PREFATA

Aceasta carte se constituie într-un efort de prezentare a unor aspecte aferente vastei problematici a microarhitecturilor de procesare a informatiei, din cel putin trei puncte de vedere "intersectate": formativ, informativ si aplicativ. Consideram ca necesitatea unei asemenea carti este imediata în contextul societatii informationale spre care ne îndreptam. În particular pentru România, cu atât mai mult este de dorit o întelegere profunda a faptului ca - printre altele - o viata mai buna este posibila numai printr-o maturizare a domeniului "electronicii si calculatoarelor" si, în consecinta, a cresterii complexitatii proiectelor ingineresti autohtone. Acesta este si mesajul esential al cartii noastre. Subliniem acest lucru pentru ca, din pacate, mai exista unele opinii care considera ca "aplicatiile hardware" performante nu mai sunt posibile în România de azi, uitând ca, daca singurele "output - uri" vizibile utilizatorului aplicatiei sunt reprezentate de monitorul video sau de imprimanta, complexitatea inginereasca este fatalmente extrem de limitata si ca urmare, beneficiul social, la fel. Iar apoi, nu este rau sa fie clar pentru toata lumea, faptul ca nici un algoritm sau program nu ruleaza prin eforturile magice ale vreunui dracusor fantomatic, chiar daca, vrând - nevrând, nociva disjunctie "hardware-software" ("tehnologie-concept") tine, câteodata, locul naturalei fuziuni. Ca si contra-balansare la acest gen de "opinii" pesimiste, subliniem faptul ca tehnologiiile de proiectare automata în microelectronica ("EDA Techniques") cu performante practic sincronizate cu nivelul mondial, sunt accesibile acum pe scara larga si la noi. Ele se constituie în suportul tehnologic principal al arhitecturilor si aplicatiilor dedicate ("embedded"), atât de necesare si diverse azi, când vorbim de informatica si electronica "domestica", personalizarea aplicatiilor, control automat si robotica etc.

Pe scurt, lucrarea este structurata astfel. În capitolul 1 se prezinta elementele esentiale ale arhitecturii microcontrolerelor - instrumente importante ale dezvoltarii de arhitecturi si aplicatii dedicate. Capitolele 2 si 3 fac o trecere în revista a principalelor caracteristici hardware-software ale actualei generatii de microprocesoare, de la ierarhizarea sistemului de memorie pâna la optimizarea de cod. Capitolul 4 prezinta un studiu de caz axat pe moderna arhitectura HSA (Hatfield Superscalar Architecture). Se prezinta aici, în premiera la noi în tara credem noi, principiile moderne ale optimizarii statice de cod, pe baza unor exemple sugestive. La fel, capitolul

5 prezinta succint noua arhitectura Intel pe 64 de biti IA-64, al carui prim procesor comercial se va numi "Itanium". Capitolul 6 este unul "de forta", poate cel mai drag autorilor, în care, pe baza unei extrem de laborioase cercetari si cugetari bibliografice, s-a încearcat predictionarea caracteristicilor hard-soft ale urmatoarei generatii arhitecturale de microprocesoare comerciale. Marturisim acum ca, scriindu-l, ne-am "înfierbântat" copios, încântati de rolul nostru temporar de…prezicatori. Capitolul urmator, al 7-lea, are drept scop justificarea faptului ca arhitectura calculatoarelor se constituie într-un domeniu viu si fascinant, posibil a fi investigat de fiecare dintre noi. Va recomandam sa-l cititi si apoi sa încercati sa va construiti propriul dvs. simulator, dedicat unei anumite probleme. În capitolul 8 am prezentat câteva aspecte privind arhitectura sistemelor multiprocesor, incluzând modelele de performanta si problemele de coerenta si sincronizare implicate. Capitolele 9 si 10 sunt dedicate propunerii si rezolvarii de probleme. Consideram ca numai prin solutionarea sistematica de probleme diverse si fecunde se poate aprofunda o asemenea disciplina tehnica de vârf, prin excelenta aplicativa.

Modernitatea si caracterul pragmatic al lucrarii sunt întarite prin acompanierea textului scris de un compact disc care contine aplicatii utile, servind ca suport concret pentru cei care vor dori sa treaca dincolo de o simpla informare. Pe CD-ul atasat se afla la dispozitia cititorului o multime de instrumente software originale, dezvoltate în mare parte de catre grupul nostru de cercetare, constituind tot atâtea posibilitati de investigare interactiva a microarhitecturilor avansate. Asteptam cu interes sugestiile si observatiile cititorilor, pentru care le multumim anticipat, pe adresele noastre de e-mail: vintan@jupiter.sibiu.ro si aflorea@vectra.sibiu.ro.

Aceasta lucrare nu ar fi putut sa apara fara eforturile admirabile ale d-lui prof.dr.ing. Gh. Toacse - initiatorului si contractorul programului de cooperare academica TEMPUS JEP AC -13559/98 "RESUME", prin care s-a finantat aparitia cartii. Îi multumim domnului profesor si pe aceasta cale pentru sprijinul acordat precum si pentru faptul de a ne fi transmis câte ceva din viziunea sa profesionala care intergreaza atât de firesc tehnologia cu arhitectura. Multumirile autorilor se îndreapta si asupra colegilor din Catedra de Calculatoare a Facultatii de Inginerie din Sibiu, pentru microclimatul profesional propice scrierii acestei carti. În final întreaga noastra gratitudine se cuvine familiilor noastre, pentru sprijinul generos pe care ni l-au acordat dintotdeauna. Le promitem ca pe viitorul apropiat vom încerca sa ne angajam mai putin în asemenea "proiecte" si le vom acorda mai mult timp, spre bucuria noastra si a lor.

Sibiu, decembrie 2000 *Autorii*

CUPRINS

1. ARHITECTURA MICROCONTROLERELOR

1.1. INTRODUCERE ÎN PROBLEMATICA

Capitolul de fata reprezinta o descriere a familiei de microcontrollere pe 8 biti, bazate pe arhitectura 80C51, realizate de firma Philips Semiconductors, precum si a altor componente furnizate de catre respectivul producator. Un microcontroller este un microprocesor destinat în general controlului unor procese industriale care contine memorii si diverse porturi de I/O integrate pe acelasi cip. În continuare se vor prezenta modurile de adresare, setul de instructiuni, partajarea memoriei s.a. în cadrul familiei de microcontrolle 80C51.

Microcontrollere derivate contin si o interfata seriala I^2C (magistrala de interconectare a circuitelor integrate), care permite conectarea cu usurinta la peste alte 100 de componente integrate, sporind capacitatea si functionalitatea microsistemului realizat. Pentru aplicatii industriale si automate, microcontrollerele sunt însotite de alta o magistrala seriala de control (Control Area Network - CAN).

Familia de microcontrollere pe 16 biti, 90CXXX se bazeaza pe arhitectura Motorola 68000. În timp ce microcontrollerele sunt pe 16 biti în exterior, în interior unitatea centrala a arhitecturii 68000 este pe 32 de biti. Acest fapt confera utilizatorului o putere de procesare mai mare, în conditiile cresterii necesitatilor de proiectare, trecând de la microcontrollere pe 8 biti la cele pe 16 biti. Microcontrollerele pe 16 biti ai firmei Philips Semiconductors sunt compatibile software cu codul procesorului Motorola 68000.

1.2. ARHITECTURA FAMILIEI 80C51

1.2.1. ORGANIZAREA MEMORIEI

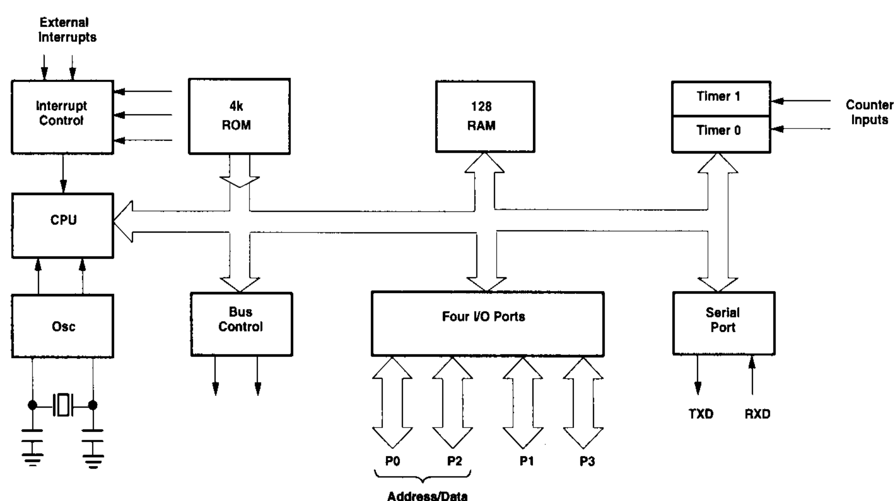


Figura 1.1. Schema bloc a microprocesoarelor 80C51

Toate procesoarele 80C51 au spatii de adrese separate pentru instructiuni si date implementând deci o arhitectura de tip Harvard a memoriei (vezi figura 1.1). Accesarea zonei de date se face pe o magistrala de 8 biti, data citita putând fi rapid memorata si manipulata de catre registrii pe 8 biti ai CPU. Memoria program este de tip ROM sau EPROM si poate avea capacitati de pâna la 64ko. La dispozitivele 80C51, cei mai putin semnificativi 4ko de memorie sunt implementati în cip. Memoria de date este de tip RAM. Cei mai putin semnificativi 128 octeti ai memoriei de date sunt implantati în cip, restul de pâna la 64ko regasindu-se extern pe placa.

Memoria program

Figura 1.2 ilustreaza harta memoriei program - partea cea mai putin semnificativa. Dupa resetarea sistemului, CPU (unitatea centrala de procesare) începe executia de la adresa 0000H, în conformitate cu initializarea PC-ului. Primii trei octeti ai Memoriei Program pot codifica de exemplu, o instructiune de salt neconditionat (*JUMP* <*Adresa*>)

reprezentând prima instructiune care se executa imediat dupa initializare. De fapt are loc un salt la adresa de început a **programului monitor** – program ce realizeaza verificarea configuratiei hardware a microsistemului, teste de memorie, interfata cu utilizatorul, etc.

Fiecarei întreruperi îi este asignata o locatie fixa în memoria program. Întreruperea determina CPU sa execute salt la locatia respectiva, unde începe executia rutinei de serviciu (tratare a întreruperii). Zona de program aferenta rutinelor de tratare a întreruperii se împarte în intervale de 8 octeti: 0003H - pentru întreruperea externa 0, 000BH - pentru circuitul Timer 0 (numarator), 0013H - pentru întreruperea externa 1, 001BH - pentru circuitul Timer 1 etc. Daca o rutina de serviciu este suficient de scurta, ea poate fi inclusa în interiorul unui astfel de interval de 8 octeti. Rutinele mai lungi de opt octeti vor folosi în general o instructiune de salt codificata pe maximum trei octeti pentru a nu altera zona aferenta unei alte întreruperi active.

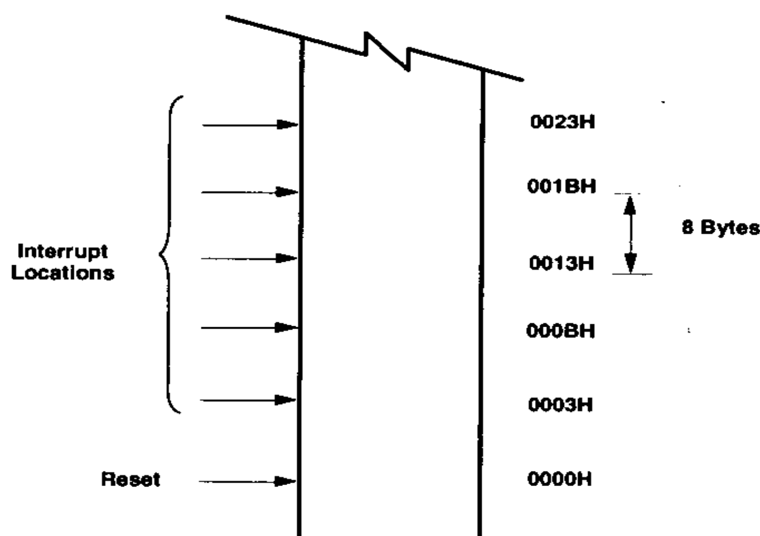


Figura 1.2. Memoria program la procesoarele 80C51

Cei mai semnificativi 4 ko ai memoriei program pot fi implementati fie în cipul ROM intern fie în memoria ROM externa. Selectia se face prin conectarea pinului $\overline{EA}$ la tensiunea de alimentare (V_{cc}) sau la masa (V_{ss}). Daca $\overline{EA}$ este legat la V_{cc}, accesele de citire din zona de memorie program cuprinsa între 0000H si 0FFFH sunt directionate spre memoria ROM – intern implementata. Accesele de citire din zona de memorie program de la adresa 1000H la FFFFH sunt îndreptate spre memoria ROM externa. Daca

$\overline{EA}$ se conecteaza la masa atunci toate citirile din memoria program sunt directionate spre memoria externa ROM.

Figura 1.3 reprezinta configuratia hardware pentru executia unui program stocat în memoria program externa. Se observa 16 linii de intrare / iesire (porturile 0 si 2) având functii de magistrala dedicata citirii codului (datei) din memoria program externa. Portul 0 serveste la multiplexarea magistralei de date / adresa. Aceasta multiplexare implica desigur o scadere a vitezei de lucru cu memoria si este datorata unor constrângeri tehnologice legate de numarul de pini ai microcontrollerului. În cadrul unui ciclu de aducere (fetch) a instructiunii el emite octetul cel mai putin semnificativ al registrului Program Counter (PCL) ca si adresa si ramâne în stare de asteptare pâna la sosirea octetului de cod din memoria program. În momentul în care octetul cel mai putin semnificativ al registrului Program Counter este valid în portul 0, semnalul ALE (Address Latch Enable) strobeaza acest octet într-un latch ale carui iesiri ataca memoria. Între timp, portul 2 emite cel mai semnificativ octet al registrului Program Counter (PCH). În final, semnalul de validare a citirii ($\overline{PSEN}$) se activeaza iar EPROM-ul emite octetul de cod cerut de catre microcontroller, prin intermediul aceluiasi port P0, aflat de data aceasta pe post de magistrala de date.

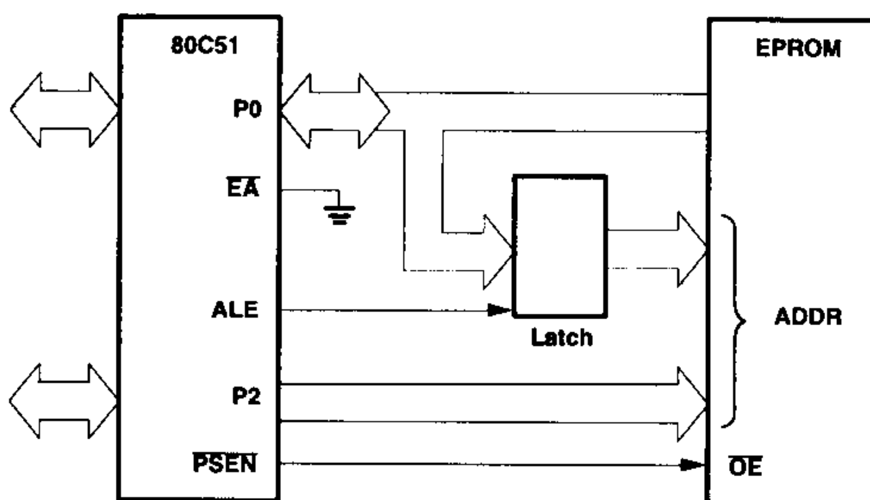


Figura 1.3. Executia programelor stocate în Memoria Externa

Adresarea memoriei program se face întotdeauna pe 16 biti, chiar daca capacitatea memoriei program fizic implementata este mai mica de 64 ko.

Executia programelor externe sacrifica doua din porturile pe 8 biti (P0 si P2) acordându-le acestora functii de adresare a memoriei program.

Memoria de date

În figura 1.4, se prezinta o configuratie hardware pentru accesarea de pâna la 2 ko de memorie RAM externa. CPU în acest caz executa instructiunile din memoria ROM interna. Portul 0 serveste ca multiplexor al magistralei de date respectiv adrese care "ataca" memoria RAM, iar cele 3 linii de intrare / iesire ale portului 2 sunt folosite la paginarea memoriei RAM (8 pagini). CPU genereaza semnalele de comanda $\overline{RD}$ si $\overline{WR}$ (validare citire respectiv scriere) necesare în timpul acceselor la memoria RAM externa.

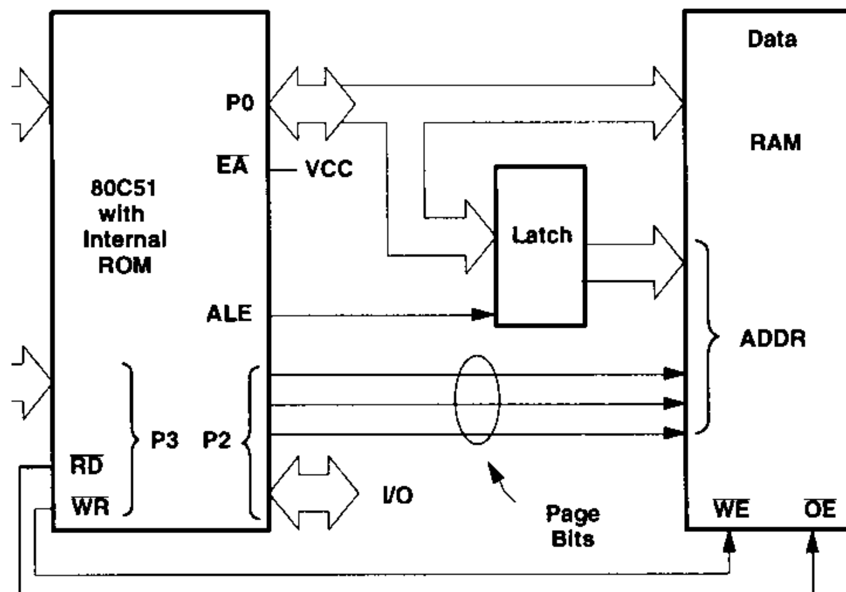


Figura 1.4. Accesarea memoriei de date externa

Adresarea memoriei de date externe poate fi facuta pe 8 sau 16 biti. Adresarea pe 8 biti este deseori folosita în conjunctie cu una sau mai multe linii de intrare / iesire pentru paginarea memoriei RAM. Adresarea pe 16 biti implica folosirea portului 2 ca emitent al octetului cel mai semnificativ de adresa, asa cum s-a mai aratat.

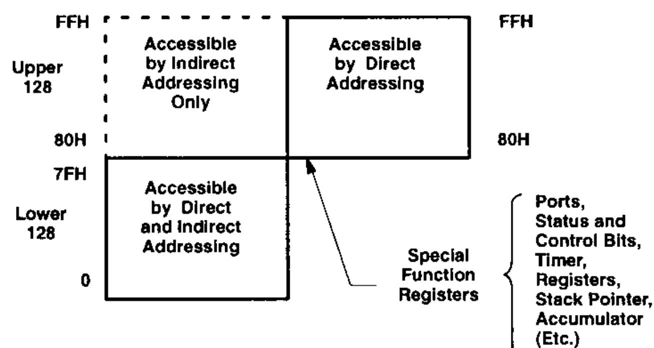


Figura 1.5. Memoria de date interna

Memoria de date interna este împartita în trei blocuri (vezi figura 1.5), referite sub numele: *cei mai putin semnificativi* 128o (inferiori), *cei mai semnificativi* 128o (superiori), si SFR (spatiu alocat registrilor cu functii speciale). Adresarea memoriei de date interne se face pe cuvinte de 1 octet rezultând un spatiu adresabil de 256o. Folosind un mic artificiu, modurile de adresare ale memoriei de date interne pot gazdui 384o si nu doar 256o cum s-ar parea la o prima vedere. Adresarea celor 128o inferiori (00 - 7FH) se poate face direct sau indirect. Adresarea celor 128o superiori (80 - FFH) se face doar prin adresare indirecta iar accesul la spatiul registrilor cu functii speciale (SFR) se face doar prin adresare directa. Rezulta ca zona de 128o superiori si spatiul SFR ocupa acelasi bloc de adrese, de la 80H la FFH, desi fizic constituie doua entitati diferite.

Figura 1.6 reflecta maparea celor 128o inferiori ai memoriei interne. Cei mai putin semnificativi 32 de octeti sunt grupati în 4 bancuri a câte 8 registri. Instructiunile programelor apeleaza acesti registri sub numele R0÷R7. Doi biti din registrul de stare program (PSW) selecteaza bancul de registri folosit. Aceasta permite o eficientizare a spatiului de cod întrucât instructiunile cu operare pe registri ocupa mai putin spatiu în memoria program decât instructiunile care folosesc adresarea directa. Urmatorii 16o, succesorii bancurilor de registri formeaza un bloc de memorie adresabil pe bit. Setul de instructiuni al microcontrollerului 80C51 cuprinde un numar mare de instructiuni avînd operanzi codificati pe un singur bit.

Maparea spatiului de memorie aferent registrilor cu functii speciale (SFR) este exemplificata în figura 1.7. De remarcat ca în spatiul alocat SFR nu toate adresele sunt ocupate. Adresele libere nu sunt implementate în cip fiind probabil rezervate pentru îmbunatatiri ulterioare ale arhitecturii. Accesele de citire la aceste locatii vor returna date aleatoare, iar accesele de scriere nu vor avea nici un efect. Dintre registrii cu functii speciale amintim: acumulatorul (A), registrul de stare program (PSW), pointerul de stiva (SP),

pointerul de date (DPTR), registrul tampon (buffer) serial de date (SBUF), registrii timer, de control, de validare întreruperi, cu prioritati de întrerupere, 4 porturi.

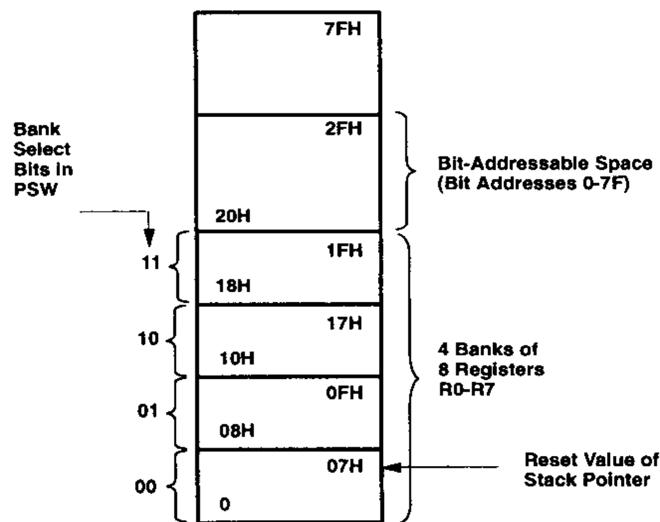


Figura 1.6. Reprezentarea celor 128 octeti inferiori ai memoriei RAM interna

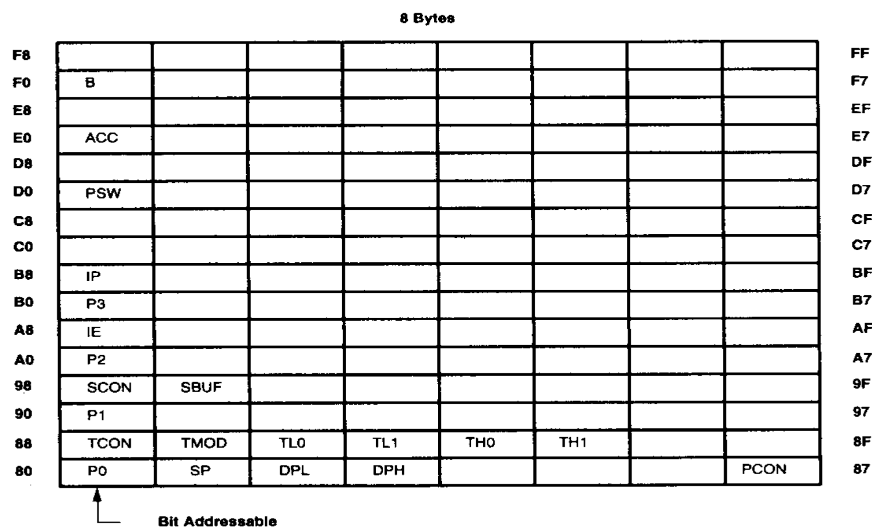


Figura 1.7. Maparea spatiului de memorie aferent registrilor cu functii speciale

Registrul de stare program (vezi figura 1.8) contine biti de stare care reflecta starea curenta a CPU. Contine bitii de transport - Carry, Auxiliary Carry, de depasire - Overflow, de paritate, doi biti de selectie ai bancului de registre si doi biti de stare la dispozitia utilizatorului. Registrul B este folosit în cadrul operatiilor de înmultire / împartire. Registrul SP este pe 8 biti si este incrementat înainte ca data sa fie memorata în timpul executiei instructiunilor PUSH sau CALL (SP - pointeaza spre ultima locatie ocupata din stiva). Acest lucru este atipic întrucât în majoritatea procesoarelor registrul SP este predecrementat la salvarea în stiva si nu preincrementat ca în acest caz. Desi stiva poate rezida oriunde în memoria RAM, registrul SP este initializat cu valoarea 07H imediat dupa semnalul Reset. Aceasta determina ca stiva sa înceapa practic de la locatia 08H. Registrul DPTR poate fi folosit ca un registru pe 16 biti sau ca doi registri independenti pe 8 biti (DPH - octetul superior al registrului DPTR si DPL - octetul inferior al registrului DPTR). Registrii pereche (TH0, TL0) si (TH1, TL1) sunt registri numaratori pe 16 biti pentru circuitele timer 0 si 1. Registrul TMOD este un registru de control si specifica modul de lucru a celor doua circuite timer. Alti registri de control sunt TCON (control al circuitelor timer), SCON (control al portului serial), PCON (control al sursei de alimentare).

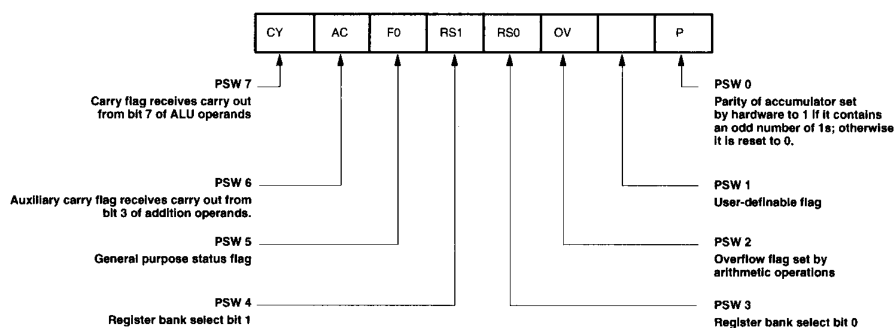


Figura 1.8. Registrul de Stare Program (PSW)

Setarea / resetarea bitilor de selectie ai celor patru bancuri de registre (PSW_3, PSW_4) se face prin metode software. De exemplu: orice instructiune care adreseaza spatiul de memorie de date 00H ÷ 1FH modifica corespunzator bitii de selectie din PSW.

1.2.2. MODURILE DE ADRESARE

Pentru aplicatii de control pe 8 biti, setul de instructiuni al microcontrollerului 80C51 a fost optimizat. Acesta permite o varietate mare de moduri rapide de adresare, pentru accesarea memoriei RAM interne, facilitând operatii pe octet asupra structurilor de date de dimensiuni reduse. Setul de instructiuni permite manevrarea directa a operanzilor la nivel de bit în sisteme logice si de control care necesita procesare booleana.

a. Adresare directa - operandul este specificat printr-un câmp de adresa pe 8 biti al instructiunii. Doar memoria RAM interna si SFR sunt adresabile direct.

Exemplu: ADD A, 7FH ;adunare în mod de adresare direct memorie

Dupa cum s-a aratat, registrul Acumulator (adresa E0H în spatiul SFR) apartine zonei de memorie SFR si poate fi adresat direct. Astfel, instructiunea de adunare devine:

ADD E0H, 7FH

si s-ar putea crede ca respectiva instructiune este codificata pe 3 octeti (1 – opcode-ul instructiunii; 2,3 – cei doi operanzi [adresele de memorie]). Totusi, respectiva instructiune este codificata pe doar 2 octeti (1 – opcode-ul instructiunii [ce include si primul operand - acumulatorul] si 2 – al doilea operand [adresa de memorie]) lucru aratat si în [1]. De fapt, arhitectura microcontrollerului fiind orientata pe acumulator (instructiunile aritmetico – logice cu doi operanzi au acumulatorul implicit ca sursa si destinatie), acesta – prin exceptie fata de ceilalti registri SFR – nu mai este necesar a fi adresat direct prin adresa E0H, fiind codificat în chiar opcode-ul instructiunii. Astfel, instructiunile aritmetico – logice cu doi operanzi în modul de adresare direct sunt codificate pe doar doi octeti în loc de trei.

b. Adresare indirecta - adresa operandului este specificata în mod indirect prin intermediul unui registru pointer. Pentru adrese pe octet registrii folositi sunt R0 sau R1 din bancul de registri selectat, sau SP (stack pointer) în cazul acesarii stivei. Pentru adrese pe doi octeti se foloseste doar registrul pointer de date (DPTR). Atât memoria RAM interna cât si cea externa sunt adresabile indirect.

Exemplu: ADD A, @R0

c. Adresarea prin registri - 3 biti din opcode-ul instructiunii specifica unul din cei 8 registri (R0÷R7) care vor fi accesati. Bancul de registri este specificat prin cei doi biti dedicati ai

registrului PSW în momentul executiei instructiunii. Instructiunile care acceseaza registrii în acest mod se numesc optimizatoare de cod întucât se elimina necesitatea unui octet de adresa (de exemplu, în modul direct, adresarea R0 ÷ R7 mai consuma un octet care în plus trebuie si adus din memoria program).

Exemplu: ADD A, R7

d. Adresarea prin instructiuni cu registrii specifici - este cazul instructiunilor care opereaza asupra registrului acumulator (A) sau pointer de date (DPTR). Nu e necesar un octet de adresa pentru operanzii respectivi, codificarea operanzilor se face în chiar opcode-ul instructiunii.

e. Adresarea prin constante imediate - folosita la încarcarea unei valori imediate într-un registru. Valorile pot fi în sistem zecimal sau hexazecimal.

Exemplu: ADD A, #127

f. Adresarea indexata - este folosita la citirea tabelelor de memorie program. Doar memoria program este adresabila indexat. Registrul DPTR sau PC retine adresa de baza a tabelului, iar registrul acumulator retine numarul intrarii în tabel. Adresarea indexata este folosita si în cazul instructiunilor de selectie de tip "*case*" din limbajele de nivel înalt. În acest caz adresa instructiunii destinatie se calculeaza ca suma dintre un pointer de baza si acumulator.

Exemplu: MOVC A, @A+DPTR

1.2.3. TIPURI DE INSTRUCTIUNI

a. Instructiuni aritmetice - sunt ilustrate în tabelul 1.1, împreuna cu modurile de adresare aferente, timpul de executie, operatia executata. Timpul de executie presupune o frecventa de ceas de 12 MHz iar instructiunile si datele se presupune ca sunt stocate în memoriile interne 80C51.

Obs. *1.* Rezultatul pe 16 biti al înmultirii dintre registrul B si acumulator (A), este depus în registrul obtinut prin concatenarea registrelor B si A.

Obs. *2.* Instructiunea DIV AB realizeaza împartirea dintre A si data din registrul B si depune câtul în registrul A si restul în B. Instructiunea DIV AB se foloseste mai putin în rutine matematice de împartire decât în conversii de baza sau operatii de deplasare (shift) - aritmetice.

Obs. *3.* Modifica flagurile din PSW în concordanta cu operatia executata, astfel: adunarea, scaderea (C, OV, AC), împartire / înmultire (C, OV) iar celelalte instructiuni doar bitul C. Totodata, toate tipurile de instructiuni (aritmetico – logice, de transfer, booleene, etc) altereaza flagurile PSW_3 si PSW_4 pentru selectia bancului corespunzator de registre din spatiul 00H ÷1FH al memoriei de date, dupa cum de altfel am mai aratat.

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION
		DIR	IND	REG	IMM	TIME (µs)
ADD A,<byte>	A = A + <byte>	X	X	X	X	1
ADDC A,<byte>	A = A + <byte> + C	X	X	X	X	1
SUBB A,<byte>	A = A – <byte> – C	X	X	X	X	1
INC A	A = A + 1	Accumulator only				1
INC <byte>	<byte> = <byte> + 1	X	X	X		1
INC DPTR	DPTR = DPTR + 1	Data Pointer only				2
DEC A	A = A – 1	Accumulator only				1
DEC <byte>	<byte> = <byte> – 1	X	X	X		1
MUL AB	B:A = B x A	ACC and B only				4
DIV AB	A = Int[A/B] B = Mod[A/B]	ACC and B only				4
DA A	Decimal Adjust	Accumulator only				1

Tabelul 1.1.

Instructiunile aritmetice aferente microcontrollerului 80C51

b. Instructiuni logice - sunt reprezentate în tabelul 1.2. Instructiunile care realizeaza operatii booleene (AND, OR, XOR, NOT) asupra operanzilor octeti, executa operatia booleana la nivel de bit. Operatiile booleene pot fi executate asupra operanzilor octeti în spatiul memoriei interne de date fara a fi necesar transferul acestor operanzi în acumulator (se salveaza timp si efort necesar salvarii în stiva).

Obs. Modifica flagul C din PSW.

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION TIME (μs)
		DIR	IND	REG	IMM	
ANL A,<byte>	A = A.AND. <byte>	X	X	X	X	1
ANL <byte>,A	<byte> = <byte> .AND.A	X				1
ANL <byte>,#data	<byte> = <byte> .AND.#data	X				2
ORL A,<byte>	A = A.OR.<byte>	X	X	X	X	1
ORL <byte>,A	<byte> = <byte> .OR.A	X				1
ORL <byte>,#data	<byte> = <byte> .OR.#data	X				2
XRL A,<byte>	A = A.XOR. <byte>	X	X	X	X	1
XRL <byte>,A	<byte> = <byte> .XOR.A	X				1
XRL <byte>,#data	<byte> = <byte> .XOR.#data	X				2
CRL A	A = 00H	Accumulator only				1
CPL A	A = .NOT.A	Accumulator only				1
RL A	Rotate ACC Left 1 bit	Accumulator only				1
RLC A	Rotate Left through Carry	Accumulator only				1
RR A	Rotate ACC Right 1 bit	Accumulator only				1
RRC A	Rotate Right through Carry	Accumulator only				1
SWAP A	Swap Nibbles in A	Accumulator only				1

Tabelul 1.2.

Instructiunile logice aferente microcontrollerului 80C51

c. Instructiuni de transfer date

c1) din memoria de date interna

Tabelul 1.3 descrie instructiunile care realizeaza transferuri de date din sau în spatiul memoriei interne. Instructiunea *MOV <dest>, <src>* permite transferuri între oricare din spatiile memoriei interne sau SFR fara a trece operanzii prin acumulator. La dispozitivele 80C51 stiva se afla implementata în cipul memoriei RAM si creste de la adrese mici la adrese mari. Instructiunea *PUSH* incrementeaza întâi SP apoi scrie octetul în stiva iar instructiunea *POP* preia vârful stivei pentru ca mai apoi sa decrementeze SP-ul. Instructiunile *PUSH* si *POP* pot folosi adresarea directa pentru identificarea octetului salvat sau restaurat, dar uzual stiva este accesata prin adresare indirecta utilizând registrul pointer de stiva SP. Stiva poate ajunge pâna în cei 128 octeti superiori ai memoriei RAM interne, daca acestia sunt implementati. Instructiunile *XCH* si *XCHD* sunt folosite la favorizarea interschimbarii datelor (reduce numarul de octeti de cod folositi si timpul de executie; daca n-ar exista aceste instructiuni ar trebui emulate prin 3 MOV-uri).

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION TIME (μs)
		DIR	IND	REG	IMM	
MOV A,<src>	A = <src>	X	X	X	X	1
MOV <dest>,A	<dest> = A	X	X	X		1
MOV <dest>,<src>	<dest> = <src>	X	X	X	X	2
MOV DPTR,#data16	DPTR = 16-bit immediate constant				X	2
PUSH <src>	INC SP:MOV"@SP",<src>	X				2
POP <dest>	MOV <dest>,"@SP":DEC SP	X				2
XCH A,<byte>	ACC and <byte> exchange data	X	X	X		1
XCHD A,@Ri	ACC and @Ri exchange low nibbles		X			1

Tabelul 1.3.

Instructiunile de transfer care acceseaza spatiul memoriei interne RAM

Exemplu:

Presupunem ca registrul R0 contine adresa 20H si acumulatorul valoarea 3FH. La locatia RAM interna 20H se afla memorata valoarea 75H. Atunci, dupa executia instructiunii: **XCH A, @R0** la locatia 20H vom avea memorata valoarea 3FH iar acumulatorul va contine valoarea 75H. Instructiunea de interschimbare prezentata poate fi înlocuita, bineînteles în mod dezavantajos ca si timp de executie, cu o secventa de trei instructiuni MOV consecutive.

i. Interschimbare folosind XCH.

```
MOV    R0, #20H
MOV    @R0, #75H      Initializare registri
MOV    A, #3FH
XCH    A, @R0         Realizare interschimbare
```

ii. Interschimbare folosind trei instructiuni MOV.

```
MOV    R0, #20H
MOV    @R0, #75H      Initializare registri
MOV    A, #3FH
MOV    30H, A         (30H) ← A; (30H) ← 3FH
MOV    A, @R0         A ← (20H); A ← 75H
MOV    @R0, 30H       (20H) ← (30H); (20H) ← 3FH
```

c2) din memoria de date externa

Lista instructiunilor care acceseaza memoria de date externa este prezentata în tabelul 1.4. Singurul mod de adresare al memoriei de date externe este cel indirect. De observat ca, în toate accesele la memoria externa de date acumulatorul este unul din operanzi (fie sursa, fie destinatia). Semnalele de citire / scriere sunt activate doar în timpul executiei instructiunii MOVX. În mod normal, aceste semnale sunt inactive si daca ele nu vor fi folosite deloc, pinii aferenti (semnalelor) sunt disponibili ca linii de intrare / iesire suplimentari.

ADDRESS WIDTH	MNEMONIC	OPERATION	EXECUTION TIME (μs)
8 bits	MOVX A,@Ri	Read external RAM @Ri	2
8 bits	MOVX @Ri,A	Write external RAM @ Ri	2
16 bits	MOVX A,@DPTR	Read external RAM @ DPTR	2
16 bits	MOVX @DPTR,A	Write external RAM @ DPTR	2

Tabelul 1.4.

Instructiunile de transfer care acceseaza spatiul memoriei externe RAM

d. Instructiuni de citire din tabele de cautare

Tabelul 1.5 reda doua instructiuni disponibile pentru citirea din tabele de cautare (lookup) din memoria program. Tabelele de cautare pot fi doar citite, nu si actualizate. Tabelele pot avea pâna la 256 intrari (de la 0 la 255). Numarul intrarii dorite este încarcat în acumulator iar adresa de început de tabel se depune în DPTR sau PC.

MNEMONIC	OPERATION	EXECUTION TIME (µs)
MOVC A,@A+DPTR	Read program memory at (A + DPTR)	2
MOVC A,@A+PC	Read program memory at (A + PC)	2

Tabelul 1.5.

Instructiuni de citire din tabele de cautare

e. Instructiuni booleene

Dispozitivele 80C51 poseda un procesor boolean complet pe un singur bit. Tabelul 1.6 ilustreaza toate instructiunile booleene existente (salturi conditionate, instructiuni de setare, stergere, OR, AND, complementare). În cazul instructiunilor de salt, adresa destinatie este specificata printr-o eticheta sau prin adresa actuala din memoria program. Salturilor pot avea loc de la -128o la +127o în memoria program relativ la primul octet care urmeaza respectiva instructiune de salt (salturi relative).

MNEMONIC		OPERATION	EXECUTION TIME (µs)
ANL	C,bit	C = C.AND.bit	2
ANL	C,/bit	C = C.AND..NOT.bit	2
ORL	C,bit	C = C.OR.bit	2
ORL	C,/bit	C = C.OR..NOT.bit	2
MOV	C,bit	C = bit	1
MOV	bit,C	bit = C	2
CLR	C	C = 0	1
CLR	bit	bit = 0	1
SETB	C	C = 1	1
SETB	bit	bit = 1	1
CPL	C	C = .NOT.C	1
CPL	bit	bit = .NOT.bit	1
JC	rel	Jump if C = 1	2
JNC	rel	Jump if C = 0	2
JB	bit,rel	Jump if bit = 1	2
JNB	bit,rel	Jump if bit = 0	2
JBC	bit,rel	Jump if bit = 1; CLR bit	2

Tabelul 1.6.

Instructiunile booleene la microcontrollerul 80C51

Exemplu:

Consideram urmatoarea functie logica ce opereaza asupra variabilelor booleene A, B, C, D, astfel:

Q = A.B + C + /D (A **and** B **or** C **or not** D)

Variabilele logice de intrare se vor conecta la circuit prin intermediul bitilor de la 0 la 3 ai portului P1. Bitul 0 al portului P3 reprezinta iesirea functiei logice. Porturile vor fi folosite dupa cum urmeaza:

Intrarea A = Bitul 0 al portului P1 (adresa 90H) – vezi figura 1.7 (Maparea spatiului de memorie aferent registrilor cu functii speciale).

Intrarea B = Bitul 1 al portului P1 (adresa 91H)

Intrarea C = Bitul 2 al portului P1 (adresa 92H)

Intrarea D = Bitul 3 al portului P1 (adresa 93H)

Iesirea Q = Bitul 0 al portului P3 (adresa B0H)

Valoarea adresei X nu este specificata si poate avea orice valoare valida din spatiul memoriei program al microcontrollerului 80C51.

Adresa	**Secventa de instructiuni**	**Observatii**
X	MOV P1, #FFH	Initializarea Portului P1
X+3	MOV C, P1.0	Preluarea intrarii A
X+5	ANL C, P1.1	A and B
X+7	ORL C, P1.2	A and B or C
X+9	ORL C, /P1.3	A and B or C or not D
X+B	MOV P3.0, C	Predarea rezultatului
X+D	SJMP X+3	Reluarea bucla

f. Instructiuni de salt

f1) neconditionat

Tabelul 1.7 prezinta instructiuni de salt neconditionat, apeluri si reveniri din subrutina / întrerupere. Adresa de salt este specificata printr-o eticheta sau o constanta pe 16 biti. Desi în tabel se afla o singura instructiune de salt *JMP addr*, în realitate distingem trei astfel de instructiuni:

- *SJMP* (adresa destinatie este un offset relativ la adresa instructiunii curente) – salt relativ la PC (utile în scrierea programelor relocabile)
- *LJMP* (adresa destinatie este o constanta pe 16 biti) – salt direct
- *AJMP* (adresa destinatie este o constanta pe 11 biti)
- Instructiunea *Call addr* substituie, de asemenea, doua instructiuni de apel:
- *LCALL* (adresa destinatie este pe 16 biti, rezultând ca subrutina se poate afla oriunde în spatiul de 64ko ai memoriei program)

- *ACALL* (formatul adresei destinatie este pe 11 biti, subrutina aflându-se în blocul de 2ko, succesor instructiunii de apel)

Programatorul specifica asamblorului adresa subrutinei fie ca eticheta, fie ca si constanta pe 16 biti. Asamblorul are desigur sarcina de a stabili adresa în formatul corect cerut de instructiune.

Diferenta dintre instructiunile *RET* (revenire din subrutina) si *RETI* (revenire din întrerupere), este aceea ca RETI anunta sistemul de control al întreruperii ca întreruperea în curs s-a încheiat. Daca nu exista nici o întrerupere în curs în momentul executiei instructiunii RETI, atunci executia celor doua instructiuni de revenire este identica si consta în preluarea celor doi octeti din vârful stivei si încarcarea lor în PC astfel încât executia programului sa continue din punctul din care a fost întrerupt. Instructiunea RETI - spre deosebire de RET - permite unei întreruperi care a încercat sa o întrerupa pe cea în curs si având acelasi nivel de prioritate, sa se starteze la finele rutinei de tratare a întreruperii curente (altfel, aceasta cerere de întrerupere nu s-ar mai lua în considerare niciodata). Cu alte cuvinte, instructiunea de revenire din întrerupere marcheaza în mod explicit finele tratarii întreruperii si permite gestionarea unor noi cereri prin resetarea bitului aferent din registrul IP (vezi în continuare, figura 1.17).

MNEMONIC	OPERATION	EXECUTION TIME (µs)
JMP addr	Jump to addr	2
JMP @A+DPTR	Jump to A + DPTR	2
CALL addr	Call subroutine at addr	2
RET	Return from subroutine	2
RETI	Return from interrupt	2
NOP	No operation	1

Tabelul 1.7.

Instructiuni de salt neconditionat la microcontrollerul 80C51

f2) conditionat

Lista instructiunilor de salt conditionat disponibile utilizatorului dispozitivelor 80C51 este redata de tabelul 1.8. Salturile sunt relative la adresa PC (urmatoare celei de salt conditionat), într-o marja de la - -128o la + 127o. Adresa de salt e specificata identic ca la celelalte instructiuni de salt. Întrucât registrul de stare program (PSW) nu contine un bit de Zero, instructiunile *JZ* si *JNZ* testeaza continutul acumulatorului (A). Instructiunile *DJNZ* (decrementeaza si executa salt daca primul operand e diferit de zero) si *CJNE* (compara operanzii si executa salt doar daca operanzii sunt diferiti) au fost introduse pentru controlul buclelor de program.

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION TIME (µs)
		DIR	IND	REG	IMM	
JZ rel	Jump if A = 0	Accumulator only				2
JNZ rel	Jump if A ≠ 0	Accumulator only				2
DJNZ <byte>,rel	Decrement and jump if not zero	X		X		2
CJNE A,<byte>,rel	Jump if A ≠ <byte>	X			X	2
CJNE <byte>,#data,rel	Jump if <byte> ≠ #data		X	X		2

Tabelul 1.8.

Instructiuni de salt conditionat la microcontrollerul 80C51

<u>Exemplu:</u>

Se considera urmatoarea secventa care adauga o întârziere într-un program, acolo unde este inserata. Registrii R0, R1 si R2 reprezinta contoarele celor 3 bucle existente. Portul P1 este folosit pe post de numarator binar. Valoarea adresei X nu este specificata si poate avea orice valoare valida din spatiul memoriei program al microcontrollerului 80C51.

Adresa	Secventa de instructiuni	Observatii
X	INC P1	Se incrementeaza numaratorul binar
X+2	MOV R0, #02H	Seteaza prima constanta de întârziere
X+4	MOV R1, #FFH	Seteaza a doua constanta de întârziere
X+6	MOV R2, #FFH	Seteaza a treia constanta de întârziere
X+8	DJNZ R2, X+8	Decremeteaza R2 si executa salt la adresa specificata daca R2 ≠ 0
X+A	DJNZ R1, X+4	Decremeteaza R1 si executa salt la adresa specificata daca R1 ≠ 0
X+C	DJNZ R0, X+2	Decremeteaza R0 si executa salt la adresa specificata daca R0 ≠ 0
X+E	SJMP X	Reluarea bucla

1.2.4. ARHITECTURA INTERNA

Figura 1.9 prezinta schema bloc de principiu a microcontrollerelor 80C51. Toate resursele interne sunt centrate în jurul unei magistrale care permite schimbul de date practic între toate modulele componente (ROM, RAM, porturi I/O, ACC, SFR, SP etc.). Astfel de exemplu, în cazul unei operatii de adunare operanzii sursa sunt înscrisi în registrii temporari TMP1,2 (invizibili pentru programator) iar rezultatul este depus pe magistrala centrala de unde apoi este înscris în registrul destinatie (de obicei

în acumulator). Memoriile ROM si RAM sunt adresate prin registri de adrese special dedicati, desigur invizibili pentru programator. Toate resursele sunt comandate de catre o unitate de control care are relul de a genera secventiat în timp toate semnalele de comanda interne sau externe necesare desfasurarii operatiilor efectuate de catre microcontroller (aducere instructiuni / date, scriere rezultate, decodificari instructiuni, achitare întreruperi etc.). Registrul de instructiuni, destinatia implicita a oricarui ciclu de aducere instructiune, este inclus si el în aceasta unitate de comanda.

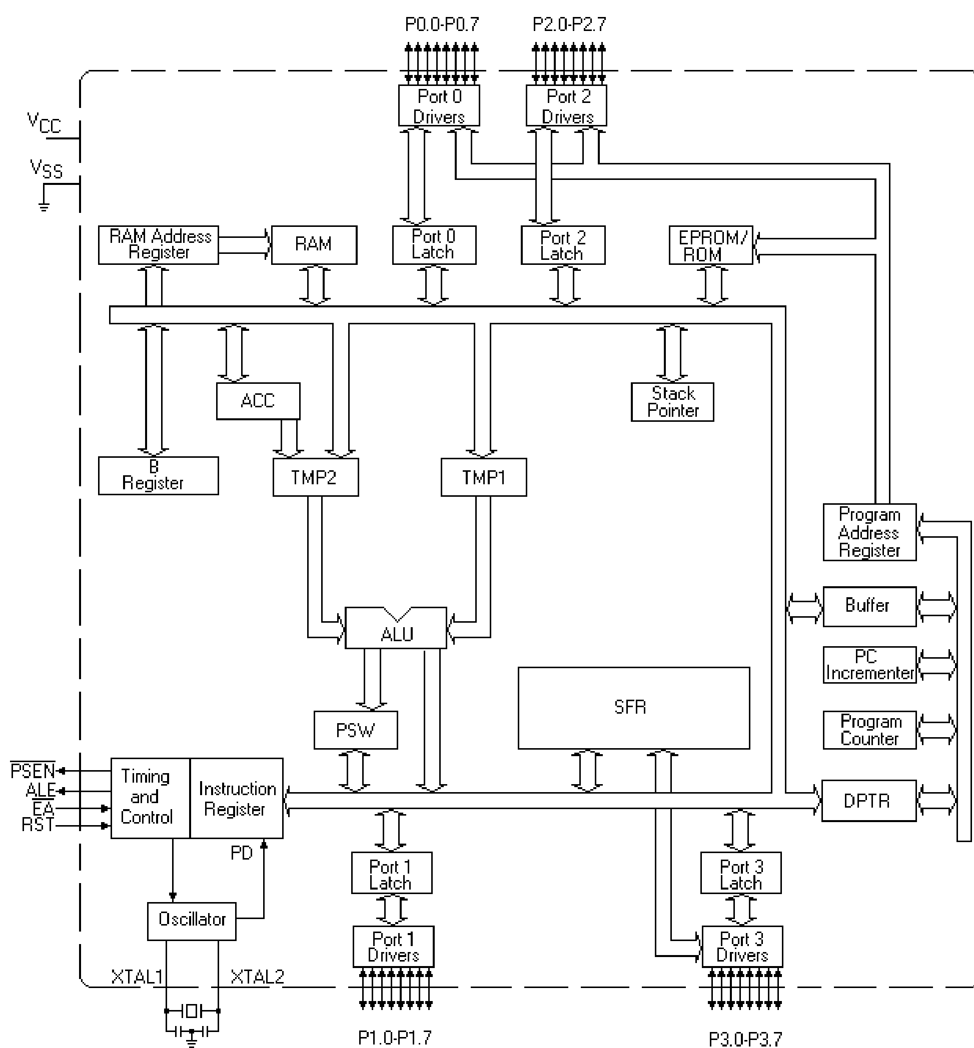


Figura 1.9. Arhitectura 80C51. Schema interna

Sursa de ceas a unitatii centrale

Toate microcontrollerele familiei 80C51 au încorporate în cip un oscilator (circuit basculant astabil), care poate fi folosit daca se doreste, ca sursa de semnal de ceas pentru CPU. În acest sens, se conecteaza cristalul de cuart sau ceramica între pinii XTAL1 si XTAL2 ai microcontrollerului, si capacitatile condensatorilor la masa (vezi figura 1.10). Figura 1.11 contine exemple de utilizare si a semnalelor de ceas extern pentru microcontroller. La dispozitivele NMOS, semnalele de la pinul XTAL2 devine generatorul de ceas intern. La dispozitivele CMOS, semnalul primit la pinul XTAL1 reprezinta sursa de ceas CPU.

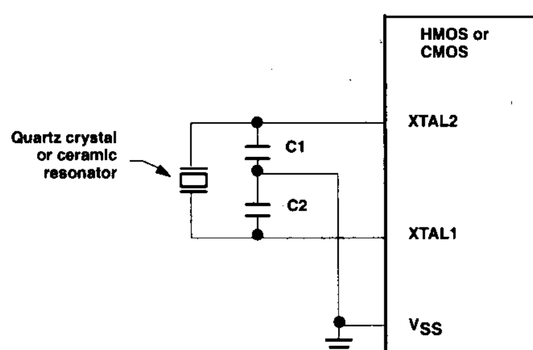


Figura 1.10. Utilizarea unui oscilator implantat în cip drept sursa de ceas

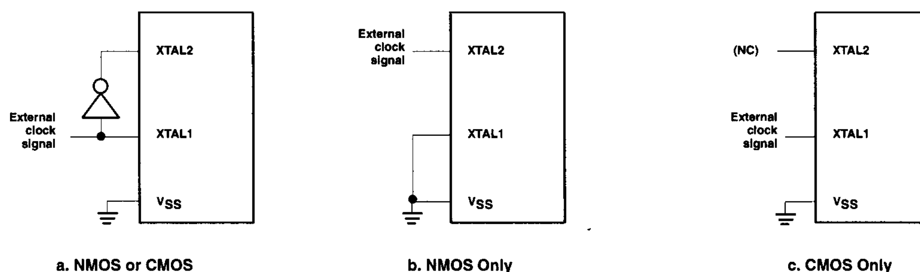


Figura 1.11. Folosirea unei surse de ceas externe pentru CPU

Accesarea memoriei externe

Distingem doua tipuri de accese la memoria externa: accese la memoria program externa (vezi figura 1.12) si accese la memoria de date externa (vezi figurile 13 si 14). Orice ciclu de acces se constituie dintr-o secventa de 6 stari, fiecare împartita în 2 perioade de tact (P1, P2).

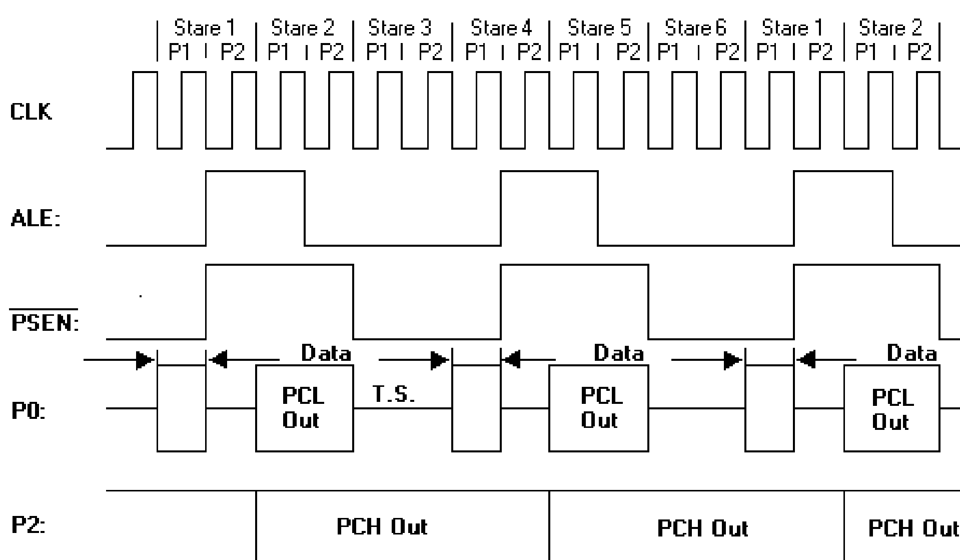


Figura 1.12. Extragerea instructiunilor din memoria program externa

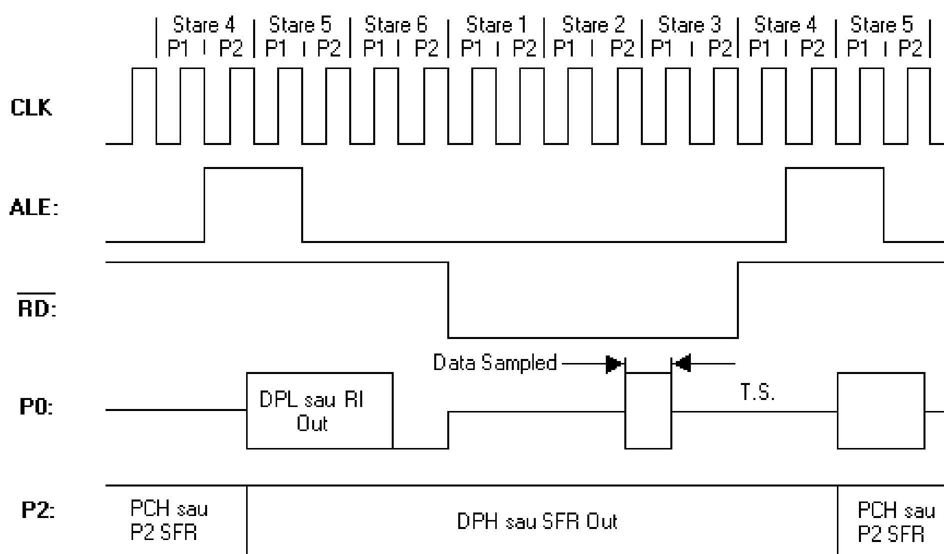


Figura 1.13. Ciclul de citire din memoria de date externa

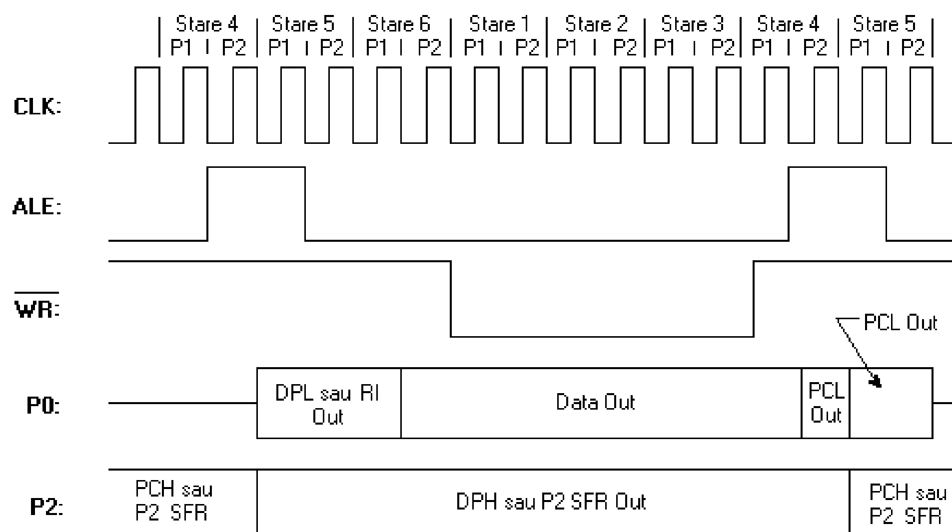


Figura 1.14. Ciclul de scriere în memoria de date externa

Semnalul de validare a citirii instructiunilor din memoria program este $\overline{\text{PSEN}}$ (program store enable). Instructiunea se citeste efectiv sincron cu un front al ceasului, în ultima parte a intervalului în care semnalul $\overline{\text{PSEN}}$ este activ, perioada în care portul P0 îndeplineste functia de magistrala de intrare date si nu de magistrala de adrese ca pâna în acest interval. Accesele la memoria de date externa folosesc semnalele $\overline{\text{RD}}$ sau $\overline{\text{WR}}$ (functii alternate ale pinilor 6 si 7 ai portului P3) pentru a valida citirea / scrierea datelor. Si în acest caz, perioada de activare a acestor doua semnale determina schimbarea functiei portului P0 din port de adrese în port de date (prin multiplexare). Desigur ca aceasta multiplexare în timp a functiilor portului P0 (adrese – date) are repercursiuni negative asupra vitezei de transfer a microcontrollerului.

Adresarea memoriei externe de program se face întotdeauna pe 16 biti, în timp ce, memoria externa de date poate fi adresata fie pe 16 biti (**MOVX A, @DPTR**) fie pe 8 biti (**MOVX A, @Ri**). La adresarea pe 16 biti, octetul superior de adresa este furnizat de portul P2 si este retinut de acesta pe toata durata ciclului de citire sau scriere. În cazul adresarii pe 8 biti, continutul portului P2 ramâne disponibil la pinii acestuia pe toata durata ciclului de citire / scriere a memoriei, una sau mai multe din liniile de intrare / iesire fiind folosite în conjunctie cu octetul de adresa furnizat de portul P0, facilitând paginarea memoriei. În ambele cazuri, octetul inferior de adresa e furnizat temporar de catre portul P0. Semnalul ALE (adrress latch enable) trebuie sa memoreze octetul de adresa într-un latch extern, întrucât în

continuare el îsi va schimba functia în magistrala de date. Octetul de adresa devine valid la tranzitia negativa a semnalului ALE, când va fi memorat în acel registru extern. În cazul ciclului de scriere, octetul de date ce va fi scris va fi disponibil în portul P0 înainte de activarea semnalului $\overline{WR}$ si ramâne astfel pâna la dezactivarea respectivului semnal. În ciclul de citire, octetul de date citit este acceptat în portul P0 chiar înainte ca semnalul $\overline{RD}$ sa fie dezactivat.

Reamintim ca pentru accesarea memoriei program externe este necesara cel putin una din conditiile: **1** – semnalul $\overline{EA}$ este activ sau **2** – registrul PC contine o valoare mai mare decât 0FFFH. Când CPU executa programe nesituate în memoria program externa, toti cei 8 biti ai portului P2 au functii dedicate de iesire si nu pot fi folosite drept linii de intrare / iesire. În timpul extragerii de instructiuni din memoria program externa, portul P2 va contine octetul superior al PC.

Structura de întreruperi

Microcontrollerele din familia 80C51 precum si cele realizate, folosind sau nu, circuite ROM sau EPROM au cinci surse de întrerupere: 2 întreruperi externe, 2 întreruperi de timer si întreruperea pe port serial (vezi figura 1.15).

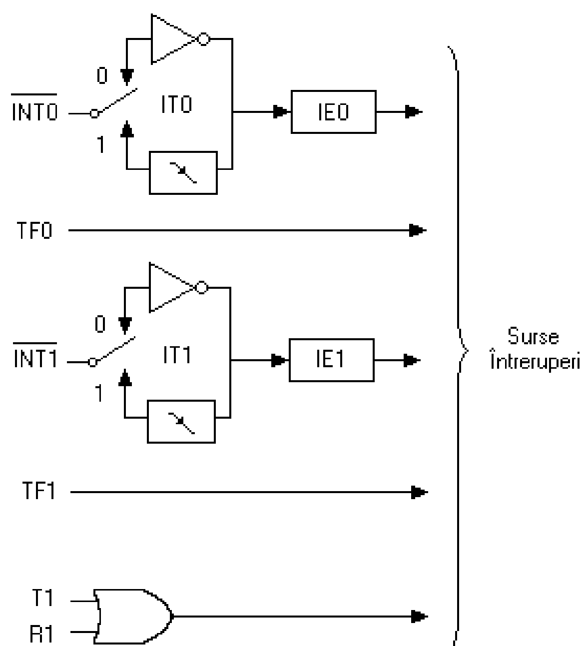


Figura 1.15. Sursele de întrerupere aferente microcontrollerului 80C51

Cele cinci surse de întrerupere sunt interne si respectiv externe. Cele 3 întreruperi endogene sunt prezentate succint în continuare. Prima ar fi cea de emisie-receptie seriala, adica P3.0 = **R_xD** (întrerupere la receptia seriala de date, deci buffer de receptie “plin”) sau P3.1 = **T_xD** (întrerupere la transmisia seriala de date, deci buffer de emisie “gol”). Diferentierea între întreruperea de emisie si cea de receptie se face doar în cadrul rutinei de tratare prin examinarea unui registru de control care specifica explicit cauza întreruperii (bit TI=1 sau/si bit RI=1). În general, în caz de conflict, se da prioritate întreruperii de receptie. Celelalte 2 întreruperi interne ar fi cele provocate de timerele interne comandate cu ceas prin pinii: P3.4 = T0 (întreruperea de timer 0 - depasire) si P3.5 = T1 (întreruperea de timer 1 - depasire). Celelalte 2 întreruperi sunt de natura exogena si anume: pe pinul P3.2 = $\overline{\text{INT0}}$ (întreruperea externa 0), pe pinul P3.3 = $\overline{\text{INT1}}$ (întreruperea externa 1).

Validarea sau invalidarea surselor de întrerupere poate fi facuta individual prin setarea sau stergerea unui bit în registrul IE (interrupt enable) din SFR. Registrul respectiv contine, de asemenea, un bit de dezactivare globala care sters, poate dezactiva toate sursele de întrerupere în acel moment (vezi figura 1.16).

(MSB) (LSB)

$\overline{EA}$	X	X	ES	ET1	EX1	ET0	EX0

Symbol	Position	Function
$\overline{EA}$	IE.7	Disables all interrupts. If $\overline{EA}$ = 0, no interrupt will be acknowledged. If $\overline{EA}$ = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
	IE.6	Reserved.
	IE.5	Reserved.
ES	IE.4	Enables or disables the Serial Port interrupt. If ES = 0, the Serial Port interrupt is disabled.
ET1	IE.3	Enables or disables the Timer 1 Overflow interrupt. If ET1 = 0, the Timer 1 interrupt is disabled.
EX1	IE.2	Enables or disables External Interrupt.1. If EX1 = 0, External Interrupt 1 is disabled.
ET0	IE.1	Enables or disables the Timer 0 Overflow interrupt.. If ET0 = 0, the Timer 0 interrupt is disabled.
EX0	IE.0	Enables or disables External Interrupt. 0. If EX0 = 0, External Interrupt 0 is disabled.

Figura 1.16. Registrul de validare al întreruperii (IE)

Prioritatile de întrerupere

Fiecare sursa de întrerupere poate fi în mod individual programata pe unul din cele doua nivele de prioritate existente, prin setarea sau stergerea unui bit într-unul din registrii SFR numit IP (interrupt priority) - vezi figura 1.17. Rutina aferenta unui nivel de prioritate scazut (low) poate fi întrerupta

de catre un nivel de prioritate ridicat (high), dar nu de catre un nivel de prioritate scazut. Un nivel de prioritate ridicat al unei întreruperi nu poate fi întrerupt de nici una din sursele de întrerupere, pe nici un nivel de prioritate. Daca doua întreruperi, fiecare fiind caracterizate de nivele de prioritate diferite, sunt receptionate simultan, cea cu nivelul de prioritate ridicat este deservita mai întâi. În cazul în care, cele doua întreruperi sunt la acelasi nivel de prioritate si sunt receptionate simultan, o secventa interna de interogare (polling) va determina care întrerupere va fi deservita prioritar. Astfel, în cadrul fiecarui nivel de prioritate (0 sau 1) exista o structura de prioritati secundara, determinata de secventa de interogare (polling), dupa cum urmeaza (vezi tabelul 1.9):

Sursa	**Prioritatea în cadrul nivelului**
IE0	Cea mai ridicata (prioritara)
TF0	.
IE1	.
TF1	.
RI+TI	Cea mai joasa (mai putin prioritara)

Tabelul 1.9.

Structura secundara de prioritati de întrerupere

IE1 / IE0 reprezinta al doilea / al patrulea bit al registrului TCON (registru de control al circuitelor timer). Sunt setati de hardware la detectia unei întreruperi externe. Sunt resetati dupa tratarea întreruperii.

TF1 / TF0 – reprezinta cel mai semnificativ / al saselea bit al registrului TCON. Sunt setati de hardware la realizarea unei operatii de depasire (overflow) de catre circuitele timer 1 sau 0. Sunt resetati prin hardware când se trece la executia rutinei de tratare a întreruperii.

TI reprezinta flagul de întrerupere pe transmisie de date. Este setat de hardware la sfârsitul transmisiei celui de-al 8-lea bit de date (buffer emisie “gol”), în modul 0 de lucru al portului serial sau la începutul transmisiei bitului de STOP în celelalte moduri de lucru, în orice transmisie seriala. Trebuie resetat prin software. RI reprezinta flagul de întrerupere pe receptie de date si la fel ca TI apartin registrului SCON (registrul de control al portului serial). Este setat de hardware la sfârsitul receptionarii celui de-al 8-lea bit de date (buffer receptie “plin”), în modul 0 de lucru al portului serial sau la jumatatea trnsmisiei bitului de STOP în celelalte moduri de lucru, în orice receptie seriala. Trebuie resetat prin software.

Exemplu: Daca flagul de validare a întreruperii este setat pe 1 în registrul IE (interrupt enable), sistemul de întreruperi genereaza un apel LCALL la locatia corespunzatoare în memoria program dupa activarea întreruperii si doar daca alte conditii nu inhiba întreruperea. Exista câteva conditii de blocare a unei întreruperi dintre care o amintim pe aceea ca o întrerupere de prioritate mai mare sau egala se afla în progres în acel moment. Instructiunea LCALL, generata practic prin hardware, determina depunerea continutului registrului PC (program counter) pe stiva si încarcarea registrului PC cu adresa de început a rutinei de serviciu (tratare a întreruperii). Reamintim ca, rutinele de serviciu ale fiecarei întreruperi încep la locatii fixate în memoria program (vezi figura 1.2). Doar registrul PC este salvat automat în stiva, nu si PSW sau orice alt registru. Acest lucru permite programatorului sa decida cât timp sa aloce salvarii altor registri functie de numarul registrilor ce trebuie salvati (cei alterati de catre rutina de tratare a întreruperii). Aceasta determina reducerea timpului de raspuns al întreruperilor, programatorul actionând direct asupra acestui parametru. Ca rezultat, multe functii de întrerupere, care se regasesc în aplicatii de control, cum ar fi: complementarea, alternanta (toggling) unui pin aferent porturilor, reîncarcarea unui numarator (timer), descarcarea unui buffer serial etc. pot fi deseori realizate într-un timp mult mai scurt decât cel necesar realizarii respectivelor functii de întrerupere pe alte arhitecturi.

(MSB) (LSB)

X	X	X	PS	PT1	PX1	PT0	PX0

Symbol	Position	Function
	IP.7	Reserved.
	IP.6	Reserved.
	IP.5	Reserved.
PS	IP.4	Defines the Serial Port interrupt priority level. PS = 1 programs it to the higher priority level.
PT1	IP.3	Defines the Timer 1 interrupt priority level. PT1 = 1 programs it to the higher priority level.
PX1	IP.2	Defines the External Interrupt 1 priority level. PX1 = 1 programs it to the higher priority level.
PT0	IP.1	Enables or disables the Timer 0 interrupt priority level. PT0 = 1 programs it to the higher priority level.
PX0	IP.0	Defines the External Interrupt 0 priority level. PX0 = 1 programs it to the higher priority level.

Figura 1.17. Registrul cu nivele de prioritati de întrerupere (IP)

Simularea unui al treilea nivel de întrerupere în software

Unele aplicatii necesita mai mult decât cele doua nivele de prioritate care sunt implementate prin hardware în cip la microcontrollerele 80C51. În acest caz se realizeaza aplicatii software simple care au acelasi efect ca si un al treilea nivel de prioritate al unei întreruperi. Pentru asta, mai întâi, întreruperii care urmeaza sa aiba prioritatea mai mare decât 1 i se asigneaza prioritatea 1 în registrul IP (interrupt priority). Rutina de tratare pentru întreruperea de prioritate 1, care se presupune posibil a fi întrerupta de întreruperea cu prioritate 2, este scrisa incluzându-se si urmatorul cod:

```
        PUSH IE
        MOV IE, #MASK ;valideaza exclusiv întreruperea de “nivel” 2 în
                       IE.
        CALL LABEL
        *****************************************************
        (Executia rutinei de serviciu – aferenta întreruperii cu nivelul 1 de
        prioritate. Aceasta poate fi întrerupta de o întrerupere de nivel 2!)
        *******************************************************
        POP IE
        RET
LABEL:
        RETI
```

De îndata ce sunt cunoscute toate prioritatile de întrerupere, registrul de validare a întreruperii (IE) este redefinit pentru a dezactiva toate sursele de întrerupere mai putin cea de prioritate 2. Apoi, prin apelul CALL la eticheta LABEL, se executa instructiunea RETI, care încheie (elibereaza) întreruperea de prioritate 1, aflata în progres (RETI – spune sistemului de control al întreruperii ca întreruperea aflata în progres s-a încheiat si prin urmare permite dupa executie luarea în considerare a întreruperii de nivel 2 pe parcursul executiei rutinei de tratare aferente întreruperii de nivel 1). În acest moment, orice întrerupere de prioritate 1 care este validata poate fi deservita, însa doar întreruperea de “prioritate 2” este validata. Dupa executia rutinei de serviciu aferenta întreruperii de prioritate 2, care poate fi tratata oriunde în memoria program, are loc restaurarea registrului IE din stiva cu octetul original de validare a întreruperilor. Apoi, instructiunea RET este folosita pentru a încheia rutina de serviciu aferenta întreruperii de prioritate 1, cea initiala.

1.3. STRUCTURA INTERFETELOR DE INTRARE / IESIRE

Microcontrollerele, spre deosebire de microprocesoare, se caracterizeaza prin includerea în propriul cip a diverselor porturi de I/O seriale si paralele, a circuitelor timer, memorii, sursa de ceas interna, etc. În compensatie, structura si filosofia lor de functionare sunt mai simple, adaptate la cerintele controlului diverselor procese industriale.

Structura porturilor

Toate cele patru porturi ale microcontrollerului 80C51 sunt bidirectionale. Fiecare consta dintr-un latch (P0÷P3) – registre din spatiul SFR, un driver de iesire si un buffer de intrare. Scrierea unui 1 respectiv 0 într-un bit al oricarui port SFR (P0, P1, P2 sau P3) determina comutarea pinului de iesire al portului corespondent în stare “high” respectiv “low”. Driverele de iesire ale porturilor P0 si P2, si bufferul de intrare al portului P0 sunt folosite în accese la memoria externa. Asa cum s-a mai aratat, portul P0 emite octetul inferior de adresa necesar adresarii memoriei externe, multiplexat cu octetul de date ce va fi scris sau citit. Portul P2 emite octetul superior de adresa catre memoria externa, în cazul adresarii pe 16 biti. Altfel portul P2 indica continutul registrului din spatiul SFR. Toti pinii portului P3 sunt multifunctionali. Acestia nu sunt doar pini ai portului 3 ci servesc si la realizarea a diverse functii (vezi tabelul 1.10).

Pinii portului P3	**Functia alternativa**
P3.0	R_xD (intrare seriala a portului)
P3.1	T_xD (iesire seriala a portului)
P3.2	$\overline{INT0}$ (întreruperea externa 0)
P3.3	$\overline{INT1}$ (întreruperea externa 1)
P3.4	T0 (intrarea externa a circuitului Timer 0)
P3.5	T1 (intrarea externa a circuitului Timer 1)
P3.6	$\overline{WR}$ (semnal de strobare la scrierea octetului de date în memoria externa)
P3.7	$\overline{RD}$ (semnal de strobare la citirea datelor din memoria externa)

Tabelul 1.10.

Functiile alternative ale pinilor portului P3

Functiile alternative pot fi activate doar daca bitul din latch-ul corespondent din SFR este 1. Rolul driverelor de iesire ale porturilor P0 si P2 poate comuta între magistrala de adresa a memoriei interne (ADDR) si rol de magistrala de adresa / date în cazul acceselor la memoria externa. În timpul acceselor la memoria externa, registrul P2 din SFR ramâne nemodificat, dar în registrul P0 din SFR este înscrisa valoarea 1. Fiecare linie de I/O poate fi folosita în mod independent atât ca intrare cât si ca iesire. Porturile P0 si P2 nu pot fi folosite ca si registre de uz general de I/O atunci când sunt folosite în operatii de accesare a memoriei externe. Toate latch-urile microcontrollerului 80C51 sunt initializate cu valoarea 0FFH de catre functia Reset. Daca într-un latch al unui port se scrie ulterior un 0, portul poate fi reconfigurat ca intrare prin scrierea unui 1 în latchul portului respectiv.

Scrierea în porturi e ilustrata în figura 1.18 fiind similara cu accesele la memorii prezentate anterior.

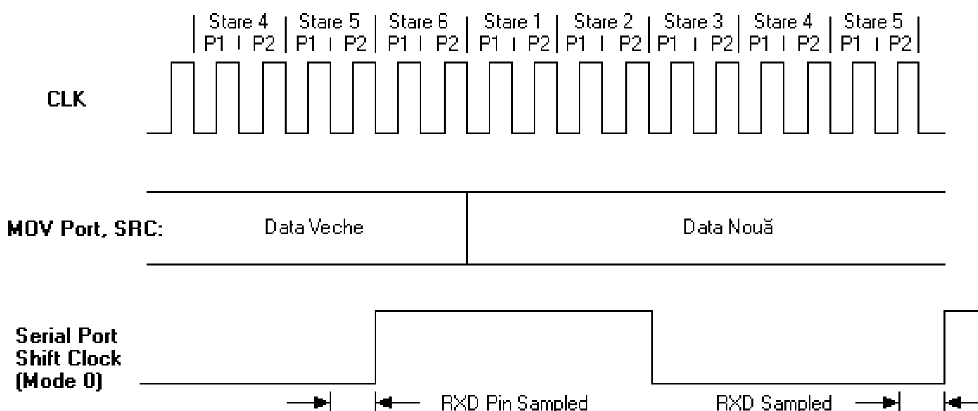


Figura 1.18. Scrierea în porturi

În executia unei instructiuni care modifica valoarea într-un latch al porturilor, noua valoare e memorata în latch în timpul fazei a doua din starea sase a ciclului final al instructiunii (S6P2). Totusi, continutul latchurilor sunt disponibile la bufferele lor de iesire doar în timpul fazei întâi a perioadei de tact iar în timpul fazei a doua bufferul de iesire retine valoarea respectiva. În consecinta, noua valoare în latchul portului nu va aparea la pinul de iesire pâna în urmatoarea faza P1 din ciclul masina urmator scrierii în port (S1P1).

Dintre instructiunile care citesc un port distingem instructiuni care citesc portul propriu zis si respectiv instructiuni care citesc pinii aferenti portului. Instructiunile care citesc porturi se caracterizeaza prin faptul ca

citesc o valoare, o modifica posibil, si apoi o rescriu în port. Operandul destinatie poate fi un port sau un bit al portului respectiv. Aceste instructiuni se numesc instructiuni "**citeste – modifica – scrie**" (read-modify-write). În continuare prezentam câteva astfel de instructiuni:

Instructiunea	**Operatia executata**
ANL P1, A	P1 SI LOGIC A (Acumulatorul)
ORL P2, A	P2 SAU LOGIC A
XRL P3, A	P3 XOR LOGIC A
JBC P1.1, Label	Daca P1.1 = 1 executa salt si reseteaza bitul
CPL P3.0	Complementeaza respectivul bit
INC P2	Incrementeaza latchul portului P2
DEC P2	Decrementeaza latchul portului P2
DJNZ P3, Label	Decrementeaza P3 si executa salt daca P3<>0
MOV Px.y, C	Transfera bitul Carry la bitul y al portului x
CLR Px.y	Reseteaza bitul y al portului x
SET Px.y	Seteaza bitul y al portului x

Tabelul 1.1.

Instructiuni de scriere în porturi

Desi nu sunt evidente, ultimele trei instructiuni sunt de tipul "citeste – modifica – scrie". Acestea citesc octetul de date al portului (toti cei 8 biti), modifica bitul adresat si apoi scrii noul octet în port. Motivul pentru care instructiunile de tipul "citeste – modifica – scrie" sunt directionate mai mult catre porturi decât catre pini consta în evitarea unei posibile interpretari gresite a nivelului electric al pinilor. De exemplu, se considera ca un bit al unui port este folosit la comanda bazei unui tranzistor. Când acest bit este setat pe '1' tranzistorul este pornit. Daca CPU citeste apoi acelasi bit al portului la nivel de pin, acesta va citi tensiunea de baza a tranzistorului si va fi interpretata ca 0. Citind acelasi bit din latchul aferent portului respectiv, vom avea valoarea corecta, si anume 1.

Interfata seriala standard

Portul serial este de tip "full – duplex", ceea ce înseamna ca poate emite si receptiona date simultan. Registrii de emisie – receptie ai portului serial sunt accesati prin registrul SBUF din spatiul SFR. Bufferul serial consta de fapt din doua registre separate, un buffer de emisie si unul de receptie. Când o data este depusa în SBUF, ea e directionata spre bufferul de emisie si retinuta pentru transmisie seriala. Când o data este mutata din

SBUF aceasta provine din bufferul de reeptie. Portul serial poate opera în patru moduri asincrone:

a. **Modul 0**. Intrarea si iesirea seriala se face pe linia R_xD. La iesirea T_xD vom avea linia de ceas. Sunt transmisi / receptionati 8 biti de date, începând cu cel mai putin semnificativ (LSB). Rata de transfer a datelor (exprimata în baud) este fixata la 1/12 din frecventa de oscilatie a generatorului de tact.

b. **Modul 1**. Sunt transmisi 10 biti (pe linia T_xD) sau receptionati (pe linia R_xD), în formatul asincron: un bit de start (0), 8 biti de date (cel mai putin semnificativ - primul) si un bit de stop (1). La receptie, bitul de stop e înscris în RB8, bit apartinând registrului SCON (vezi figura 1.19). Rata de transfer este variabila, functie de frecventa de tact.

c. **Modul 2**. Sunt transmisi (pe linia T_xD) sau receptionati (pe linia R_xD) 11 biti: bitul de start (0), 8 biti de date (primul LSB), un bit programabil (al 9-lea bit de date) si un bit de stop (1). La transmisia datelor, celui de-al 9-lea bit de date (bitul TB8 din SCON – vezi figura 1.19) îi poate fi asignata valoarea 0 sau 1. La receptie, cel de-al 9-lea bit este înscris în RB8 al SCON, iar bitul de stop este ignorat. Rata de transfer este programabila fie la 1/32 fie la 1/64 din frecventa de oscilatie a generatorului de tact.

d. **Modul 3**. Este identic cu modul 2 de operare, exceptând rata de transfer. În modul 3, rata de transfer este variabila. Util în comunicatia multiprocesor dupa cum se va arata în continuare.

În toate cele patru moduri transmisia este initiata de catre orice instructiune care foloseste registrul SBUF (buffer de emisie) ca destinatie. Receptia este initiata în modul 0 de catre conditiile **(RI=0) AND (REN=1)**. Receptia este initiata, în celelalte moduri, clasic pentru protocoalele asincrone, de catre sosirea bitului de start daca **REN=1**.

Registrul de stare si control al portului serial – SCON (vezi figura 1.19) contine nu doar bitii de selectie ai modului de operare ci si al 9-lea bit de date dintr-o transmisie sau receptie (TB8 si RB8), si bitii de întrerupere ai portului serial (TI si RI).

(MSB) (LSB)

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0, SM1 - specifica modul de lucru al portului serial.

SM0	SM1	Descriere	Rata de transfer
0	0	Registru de deplasare	$f_{OSC}/12$
0	1	UART pe 8 biti	variabila
1	0	UART pe 9 biti	$f_{OSC}/64$ sau $f_{OSC}/32$
1	1	UART pe 9 biti	variabila

SM2 - valideaza caracteristica de comunicatie multiprocesor in modurile 2 si 3. In aceste doua moduri, daca SM2 e setat pe 1 atunci RI nu va fi activat daca al 9-lea bit de date (RB8) este 0. In modul 1, daca SM2=1, atunci RI nu va fi activat daca nu s-a receptionat un bit de STOP valid. In modul 0 SM2 trebuie sa fie 0.

REN - Setat/resetat software pentru a valida/invalida receptia seriala.

TB8 - este al 9-lea bit de date care va fi transmis in modurile 2 si 3. Setat/resetat software.

RB8 - in modurile 2 si 3 este al 9-lea bit de date care a fost receptionat. In modul 1, daca SM2=0, RB8 este bitul de STOP care a fost receptionat. in modul 0 nu este folosit.

TI - este flagul de intrerupere pe transmisie de date. Setat de catre hardware la sfarsitul celui de-al 8-lea impuls de tact in modul 0, sau la inceputul bitului de STOP in celelalte moduri. Trebuie resetat software.

RI - este flagul de intrerupere pe receptie de date. Setat de catre hardware la sfarsitul celui de-al 8-lea impuls de tact in modul 0, sau la jumatatea impulsului de tact aferent bitului de STOP in celelalte moduri. Trebuie resetat software.

Figura 1.19. Registru de control al portului serial (SCON)

Comunicatia în sisteme multiprocesor

Modurile de operare 2 si 3 trateaza special comunicatia în sisteme multiprocesor. În aceste moduri, sunt receptionati 9 biti de date. Cel de-al 9-lea bit este memorat în RB8 al registrului SCON. Apoi urmeaza bitul de STOP. Portul poate fi programat astfel încât, la receptionarea bitului de stop, întreruperea de port serial va fi activata doar daca RB8=1. Caracteristica de comunicatie multiprocesor e validata daca bitul SM2 din SCON este setat. O modalitate de a folosi respectiva caracteristica în sisteme multiprocesor este urmatoarea:

Când un procesor master doreste sa transmita un bloc de date unuia din dispozitivele slave, acesta trimite mai întâi un cuvânt de adresa care identifica slave-ul destinatie. Un cuvânt de adresa difera de unul de date prin aceea ca al 9-lea bit este 1 în cuvântul de adresa si 0 în cel de date. Când SM2=1, nici un slave nu va fi întrerupt de catre un octet de date. Un cuvânt de adresa, totusi, va întrerupe toate dispozitivele slave, astfel încât fiecare slave sa poata examina si detecta daca cuvântul receptionat reprezinta adresa sa. Slave-ul adresat va reseta bitul SM2 si se va pregati sa receptioneze cuvântul de date. Dispozitivele slave care nu au fost adresate lasa bitii SM2 aferenti lor setati si îsi continua activitatea neperturbate, ignorând cuvântul de date. SM2 nu are nici un efect în modul 0, iar în modul 1 poate fi folosit sa verifice validitatea bitului de stop. În receptia din modul 1, daca SM2=1, întreruperea de receptie nu va fi activata daca nu se receptioneaza un bit de stop valid.

Circuite Timer/Numaratoare

Microcontrollerul 80C51 contine doua registre timere/numaratoare pe 16 biti: Timer 0 si Timer 1. Ambele pot fi configurate sa opereze atât ca circuite timer (periodizatoare) cât si ca numaratoare. Având functia de timer, registrul este incrementat cu fiecare ciclu masina. Întrucât un ciclu masina consta din 12 perioade de oscilatie ale generatorului de tact, rata de numarare este 1/12 din frecventa oscilatorului. Având functia de numarator, registrul este incrementat ca raspuns la o tranzitie din 1 în 0 a intrarii externe corespondente de la pinul T_0 sau T_1. Noua valoare numerica apare în registru în timpul fazei P1 a ciclului (S3P1) urmator celui în care s-a detectat tranzitia. Întrucât dureaza doi cicli masina (24 perioade de oscilatie) pentru a recunoaste o tranzitie din 1 în 0, rata maxima de numarare este 1/24 din frecventa oscilatorului.

Pe lânga posibilitatea de selectie între functia de timer sau numarator, circuitele Timer0 si Timer1 sunt caracterizate de patru moduri de operare. Functia de timer sau numarator este selectata cu ajutorul bitilor C/T din registrul TMOD din spatiul SFR (vezi figura 1.20). Bitii M1 si M0, apartinând aceluiasi registru TMOD, selecteaz modul de operare. Modurile 0, 1 si 2 sunt aceleasi atât pentru Timer/Numaratorul 0 cât si pentru Timer/Numaratorul 1. Modul 3 este diferit functie de circuit.

(MSB) (LSB)

Gate	C/T	M1	M0	Gate	C/T	M1	M0
Timer 1				Timer 0			

Gate - Cand este setat opreste controlul. Timerul/Numaratorul 'x' este validat doar cand pinul $\overline{INT}$x este high si pinul de control TRx este setat.

C/T - Selectorul functiei de Numarator (intrarea o reprezinta ceasul intern sistemului) sau Timer (intrarea o reprezinta pinul de intrare Tx)

M1	M0	Operatia
0	0	Timer 8048.
0	1	Timer/Numarator pe 16 biti THx si TLx sunt cascadate.
1	0	Timer/Numarator pe 8 biti cu reincarcare. THx retine valoarea care urmeaza sa fie reincarcata in TLx la fiecare depasire.
1	1	(Timer 0) TL0 este un Timer / Numarator pe 8 biti controlat de bitii standard de control ai Timerului 0. TH0 este timer pe 8 biti controlat doar de biti de control ai Timerului 1.
1	1	(Timer 1) Timerul / Numaratorul 1 este oprit.

Figura 1.20. Registrul de control al modului de functionare al circuitelor Timer (TMOD)

a. Circuitele Timer în **modul 0** se comporta precum circuitul Timer 8048, care este un numarator pe 8 biti. În acest mod registrul timer este configurat ca un registru pe 13 biti. Daca numarul existent în registru devine din toti bitii pe 1 în toti bitii pe 0, este setat flagul de întrerupere pe Timerul1 (TF1). Intrarea de numarare este validata când TR1=1 si, fie GATE=0 fie $\overline{\text{INT1}}$=1. Setând GATE pe 1 se permite timerului sa fie controlat de intrarea externa $\overline{\text{INT1}}$, facilitând masurarea perioadei de tact. TR1 este un bit de control din registrul TCON apartinând SFR (vezi figura 1.21). GATE apartine registrului SMOD. Cei 13 biti ai registrului constau din 8 biti din TH1 si 5 biti din TL1. Cei 3 biti superiori ai TL1 sunt nedeterminati si trebuie ignorati. Setarea flagului de executie (TR1) nu reseteaza continutul registrului timer.

b. **Modul 1** este identic cu modul 0, exceptând faptul ca registrul timer ruleaza cu toti cei 16 biti ai sai.

c. **Modul 2** configureaza registrul timer ca un numarator pe 8 biti (TL1) cu reîncarcare automata. Depasirea din TL1 nu doar seteaza TF1 dar si reîncarcaTL1 cu continutul lui TH1, care este presetat software. Reîncarcarea lasa TH1 nemodificat. Modul 2 opereaza în acelasi mod si asupra Timerului/Numaratorului 0.

d. În **modul 3** Timerul 1 retine numarul. Efectul este identic cu setarea TR1 pe 0. Timerul 0 în modul3 identifica pe TL0 si TH0 ca doua numaratoare separate. TL0 utilizeaza bitii de control ai Timerului 0: **C/T**, **GATE**, **TR0**, **INT0** si **TF0**. TH0 este fixat (blocat) într-o functie timer (numarare a ciclilor masina) ce are ca argumente pe TR1 si TF1 din Timerul1. Astfel, TH0 controleaza si întreruperea de Timer 1. Modul 3 este furnizat pentru aplicatii care necesita timere/numaratoare ce depasesc 8 biti. Cu Timerul 0 în modul 3 de operare, microcontrollerul 80C51 simuleaza trei timere/numaratoare. Când Timerul 0 este în modul 3, Timerul 1 poate fi pornit/oprit prin comutarea sa în/din modul 3, sau poate fi folosit de catre portul serial ca generator de rate de transfer, sau în orice aplicatie care nu necesita o întrerupere.

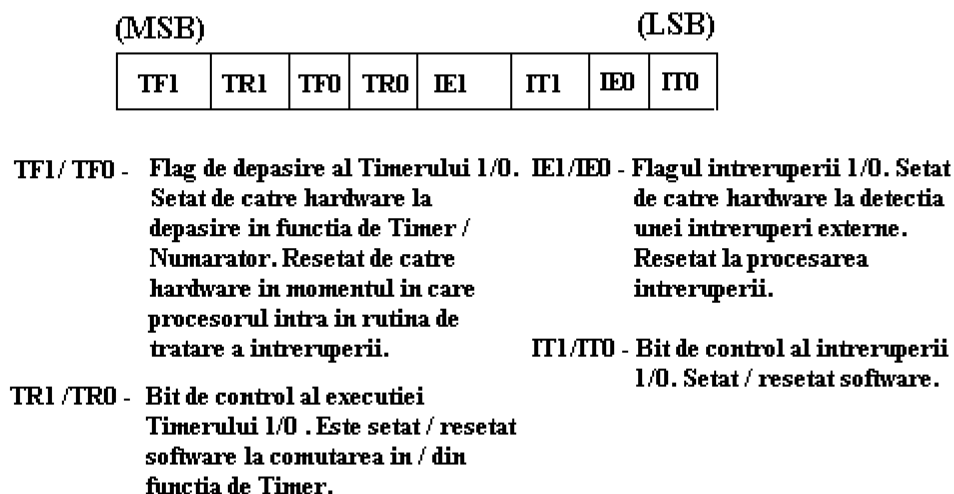


Figura 1.21. Registrul de control al cicuitelor Timer/Numarator (TCON)

1.4. MAGISTRALA DE INTERCONECTARE – I^{2}C

I^{2}C, magistrala bidirectionala pe doua fire, a fost dezvoltata de catre compania Philips, pentru eficientizarea (maximizarea performantei hardware si respectiv simplitatea circuitelor) controlului interconectarii circuitelor. Toate dispozitivele compatibile cu magistrala I^{2}C contin o interfata implementata în cip care permite tuturor dispozitivelor de acest gen sa comunice între ele prin respectiva magistrala. Acest concept de proiectare rezolva multe probleme de interfatare ce apar în proiectarea circuitelor de control digital. El se remarca prin simplitate si eficienta, caracteristici deosebit de apreciate în controlul industrial al proceselor tehnologice.

Caracteristicile magistralei de interconectare

- Necesita doar doua linii de magistrala (o linie seriala de date – SDA, si o linie seriala de ceas – SCL).
- Fiecare dispozitiv conectat la magistrala este software adresabil printr-o adresa unica si în fiecare moment exista o relatie simpla de tip *master / slave*.
- Este o magistrala *multimaster* care include detectia coliziunilor si arbitrarea acestora pentru a preveni inconsistenta datelor în cazul în care

doua sau mai multe dispozitive master initiaza simultan transferul de date.

- Pe magistrala seriala de date, pe 8 biti, au loc transferuri bidirectionale de date cu viteze pâna la 100kbit / s în mod standard sau pâna la 400kbit / s în mod rapid (fast).
- Filtrele implementate în cip elimina zgomotele datorate diafoniilor, reflexiilor, etc (spike-uri) de pe linia de date pentru pastrarea integritatii datelor.
- Numarul de circuite care pot fi conectate la aceeasi magistrala este limitat doar de capacitatea maxima a respectivei magistrale, anume de 400pF.

Avantajele proiectantului constau în facilitatea oferita de circuitele integrate interconectate prin magistrala I^2C, privind trecerea rapida de la organigrama cu blocuri functionale la prototip. Circuitele integrate (IC – Integrated Circuits) sunt conectate la magistrala I^2C fara o interfata suplimentara externa, permitând modificarea sau îmbogatirea sistemului prototip simplu prin conectarea sau deconectarea de la magistrala (sisteme de dezvoltare).

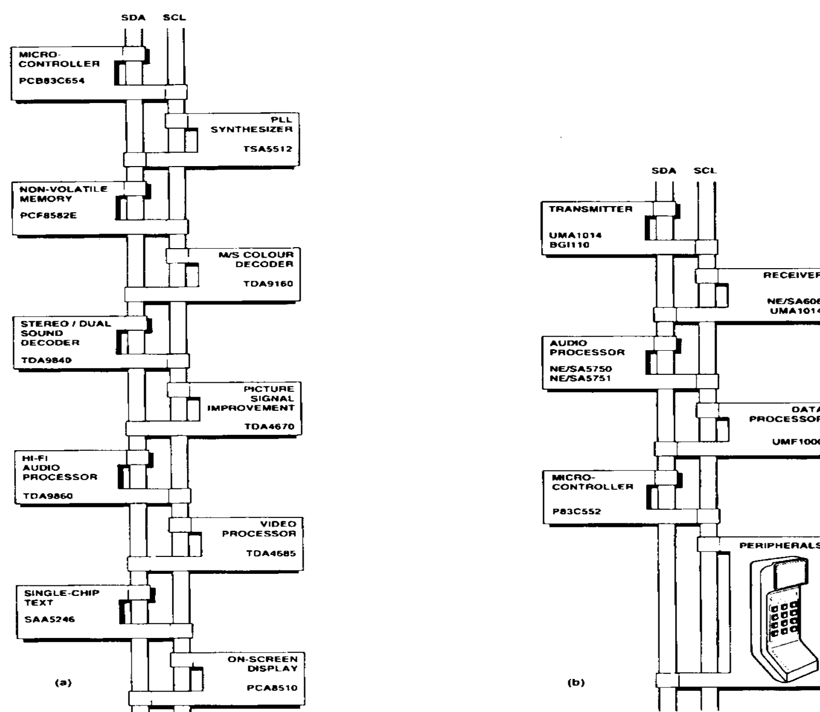


Figura 1.22. Aplicatii utilizând magistrala I^2C

Figura 1.22 ilustreaza în mod intuitiv doua aplicatii utilizând magistrala de interconectare I^2C.

Caracteristicile circuitelor integrate compatibile cu magistrala I^2C

- Blocurile functionale din organigrama corespund cu circuitele integrate actuale.
- Nu este necesara proiectarea interfetei la magistrala deoarece interfata I^2C este deja integrata în cip.
- Adresarea integrata si protocolul de transfer de date permite sistemului sa fie definit complet din punct de vedere software.
- Aceleasi tipuri de IC - uri pot fi des folosite în mai multe aplicatii diferite.
- Timpul de proiectare reduce durata procesului de familiarizare a proiectantului cu cele mai frecvent folosite blocuri functionale, reprezentate de circuitele integrate compatibile cu magistrala I^2C.
- Circuitele integrate pot fi adaugate sau înlaturate din sistem fara a afecta celelalte circuite conectate la magistrala (asadar caracteristici de modularizare si toleranta la defectari).
- Depanarea disfunctiunilor se poate realiza *pas cu pas*.
- Timpul de dezvoltare software poate fi redus prin asamblarea unor biblioteci cuprinzând module software refolosibile.

Circuitele integrate compatibile cu magistrala I^2C, de tip CMOS, ofera proiectantului proprietati speciale, atractive pentru echipamentele portabile si sistemele alimentate de baterie. Ele se caracterizeaza prin:

- Consum redus de energie electrica.
- Imunitate ridicata la zgomot.
- Suporta variatii largi de tensiune.
- Suporta variatii mari de temperatura.

Avantajele fabricantului de circuite integrate compatibile cu magistrala I^2C

- Magistrala I^2C este compusa din doua fire simple, fapt ce minimizeaza interconexiunile si fac ca circuitele integrate sa aiba un numar redus de pini.
- Protocolul de magistrala I^2C elimina necesitatea unui decodor de adrese.
- Capacitatea de multimaster a magistralei I^2C permite testarea rapida si alinierea echipamentului utilizatorului prin conexiuni externe la un computer printr-un program (eventual scris în asamblare).
- caracteristica a magistralei I^2C, apreciata atât de catre proiectanti cât si de catre fabricanti, este aceea ca, natura simpla, bazata pe doar doua fire si

capabilitatea adresarii software, fac din I^2C o platforma ideala pentru magistrala ACCESS.bus (vezi figura 1.27). Aceasta reprezinta o alternativa, din punct de vedere cost / performanta, pentru interfata RS – 232C de conectare a perifericelor la un calculator gazda printr-un conector simplu având patru pini.

Specificatii privind magistrala I^2C

Pentru aplicatii de control digital pe 8 biti, cum sunt cele care necesita microcontrollere, se stabilesc criterii principiale de proiectare, astfel:

- ⇨ Un sistem complet consta, de regula, din cel putin un microcontroller, memorii si alte dispozitive periferice cum ar fi extensiile de porturi de intrare / iesire.
- ⇨ Costul interconectarii diverselor dispozitive trebuie sa fie minim.
- ⇨ Un sistem care executa o functie de control nu necesita un transfer rapid de date.
- ⇨ Eficienta globala depinde de dispozitivele alese si de natura structurii magistralei de interconectare.

Pentru realizarea unui sistem care sa satisfaca aceste criterii, este nevoie de o structura de magistrala seriala. Desi, magistralele seriale nu au aceleasi capabilitati ca cele paralele, ele necesita mai putine fire si mai putini pini din partea circuitelor integrate ce se vor conecta. O magistrala însa, nu reprezinta numai "niste sârme" de interconectare, ci întruchipeaza toate formatele si procedurile de comunicare din interiorul sistemului. Comunicatiile între dispozitive prin intermediul magistralei I^2C trebuie realizate prin protocoale clar definite si complete, pentru a se evita toate posibilitatile de confuzie, pierderi de date si blocaje informationale. Dispozitivele rapide trebuie sa poata comunica cu dispozitivele lente. Sistemul nu trebuie sa fie dependent de dispozitivele conectate la el, altfel nu ar fi posibile eventuale modificari si îmbunatatiri. O procedura trebuie sa decida care dispozitiv va controla magistrala, si când. În cazul interconectarii dispozitivelor cu rate de ceas diferite, trebuie specificat sursa semnalului de ceas al magistralei.

Conceptul de magistrala de interconectare

Cele doua fire (SDA - date si SCL - ceas) transporta informatie între dispozitivele conectate la magistrala. Fiecare dispozitiv este caracterizat de o adresa unica – daca este microcontroller, driver LCD, memorie, interfata pentru tastatura – si pot opera fie ca *emitator* fie ca *receptor*, dependenta de functia dispozitivului. Evident ca un driver LCD este doar receptor, în timp ce memoria poate fi fie receptor fie emitator. În timpul realizarii

transferurilor de date, dispozitivele pot fi considerate ca master sau slave (vezi tabelul 1.12).

Term	Description
Transmitter	The device which sends the data to the bus
Receiver	The device which receives the data from the bus
Master	The device which initiates a transfer, generates clock signals and terminates a transfer
Slave	The device addressed by a master
Multi-master	More than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the message is not corrupted
Synchronization	Procedure to synchronize the clock signals of two or more devices

Tabelul 1.12.

Definitii privind terminologia de magistrala I^2C

Un dispozitiv este considerat *master* daca initiaza un transfer de date pe magistrala si genereaza semnalul de ceas pentru a permite transferul. În acel moment, orice dispozitiv adresat e considerat *slave*. Întrucât magistrala I^2C este de tip multimaster rezulta ca pot fi conectate la aceasta mai mult de un dispozitiv capabil de a controla magistrala. Pentru a evita "haosul" care se poate instaura în urma unor astfel de evenimente se introduce o **procedura de arbitrare**. Aceasta se bazeaza pe conexiunea **SI LOGIC** (AND) a tuturor interfetelor I^2C aferente dispozitivelor conectate la magistrala I^2C. Semnalul de ceas în timpul arbitrarii este o combinatie sincronizata a semnalelor de ceas generate de dispozitivele master folosind conexiunea **SI LOGIC** asupra liniei SCL. Fireste, generarea semnalelor de ceas pe magistrala I^2C este întotdeauna responsabilitatea dispozitivului master. Pentru transferarea datelor pe magistrala, fiecare din dispozitivele master genereaza propriul sau semnal de ceas. Acest semnal poate fi alterat doar datorita unui dispozitiv slave lent, care întârzie semnalul activ de ceas sau de catre un alt dispozitiv master când se realizeaza arbitrarea. Un dispozitiv master poate starta transferul doar daca magistrala este libera.

Transferul datelor

Pe durata unui transfer de date, apar doua situatii unice definite drept conditii de **START** si **STOP**. O tranzitie din stare HIGH în stare LOW a liniei de date (SDA), în timp ce semnalul de ceas (SCL) este în stare HIGH, indica o conditie de START. O tranzitie din LOW în HIGH a liniei de date, în timp ce semnalul de ceas ramâne în stare HIGH, defineste o conditie de STOP. Cele doua conditii de START si STOP sunt generate întotdeauna de

catre dispozitivul master. Magistrala se considera a fi ocupata dupa o conditie de START si libera dupa o conditie de STOP. Detectia celor doua conditii de catre dispozitivele conectate la magistrala este simpla daca ele încorporeaza o interfata hardware necesara.

Fiecare data depusa pe linia de date (SDA) a magistralei I^2C trebuie sa aiba lungimea de 8 biti. Numarul de octeti transmisi per transfer este nelimitat. Fiecare octet trebuie sa fie urmat de un bit de recunoastere (*Acknowledge*). Datele sunt transferate cu cel mai semnificativ bit întâi. Daca receptorul nu poate receptiona complet octetul de date, deoarece se afla în executia unui alt proces (ex: deservirea unei întreruperi), acesta retine semnalul de ceas SCL în stare LOW fortând intrarea transmitatorului în stare de asteptare (wait). Transferul de date continua de îndata ce receptorul este gata pentru a primi un alt octet de date si elibereaza semnalul de ceas. În unele cazuri, este permisa folosirea unor formate de date diferite fata de formatul I^2C – bus (de exemplu, pentru dispozitive compatibile CBUS). Un mesaj care starteaza cu o adresa CBUS poate fi terminat prin generarea unei conditii de STOP, chiar în timpul transmisiei unui octet, în acest caz, nefiind generat nici un bit de recunoastere.

Transferul de date trebuie sa cuprinda obligatoriu bitul de recunoastere. Bitul de Acknowledge este transmis de slave (vezi figura 1.23).

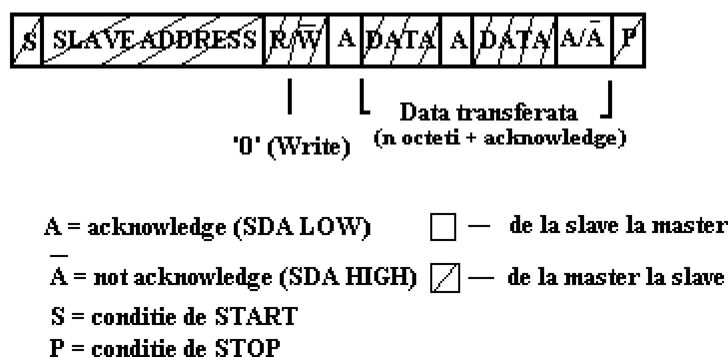


Figura 1.23. Dispozitivul Master – emitator adreseaza un Slave – receptor cu o adresa pe 7 biti

Emitatorul master elibereaza linia de date (SDA), aflata în stare HIGH, pe durata respectivului impuls de tact. Totodata, receptorul trebuie sa determine trecerea liniei de date în stare LOW. Fiecare bit de date este sincronizat cu un impuls de ceas (vezi figura 1.24).

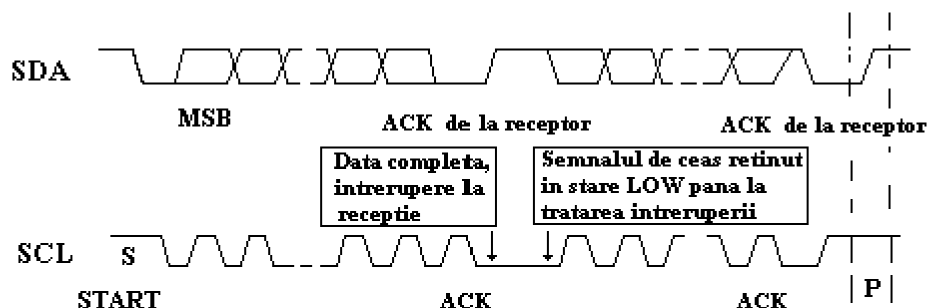


Figura 1.24. Transferul de date pe I^2C

De regula, un receptor care a fost adresat este obligat sa genereze bitul de recunoastere (acknowledge) dupa fiecare octet receptionat, exceptie facând mesajele care încep cu o adresa CBUS. Când un **slave – receptor** nu recunoaste adresa de slave (de exemplu nu poate receptiona deoarece executa o functie în timp real), linia de date trebuie lasata în stare HIGH de catre slave. Dispozitivul master poate genera atunci o conditie de STOP care va întrerupe transferul. Daca un **slave – receptor** recunoaste adresa, dar mai târziu în transfer nu mai poate receptiona nici o data, dispozitivul master trebuie sa întrerupa din nou transferul. Acest lucru este indicat de catre slave prin generarea unui bit de recunoastere negat la finele primului octet ce urmeaza. Slave-ul lasa linia de date în stare HIGH iar dispozitivul master genereaza conditia de STOP.

Daca un **master – receptor** este implicat într-un transfer, el trebuie sa semnaleze sfârsitul octetilor de date **emitatorului – slave**, prin faptul de a nu genera un bit de acknowledge dupa ultimul octet trimis de slave. Slave-ul emitator trebuie sa elibereze linia de date pentru a permite dispozitivului master sa genereze conditia de STOP.

Ca dispozitive master, de regula, sunt utilizate microcontrollere. Presupunem urmatorul exemplu, de transfer de date între doua microcontrollere conectate la magistrala I^2C (vezi figura 1.25). Considerând transferul datelor în format cu 7 biti de adresa se vor exemplifica doua situatii: una în care dispozitivul master este emitator si slave-ul receptor si alta în care dispozitivul master este receptor iar slave-ul emitator. Dupa conditia de start **S**, se transmite adresa unui slave pe 7 biti. Aceasta este urmata de un bit de directie ($R/\overline{W}$) (vezi figura 1.23). Daca acesta este ‘0’ indica o scriere de date (WRITE) iar succesiunea de mesaje este urmatoarea:

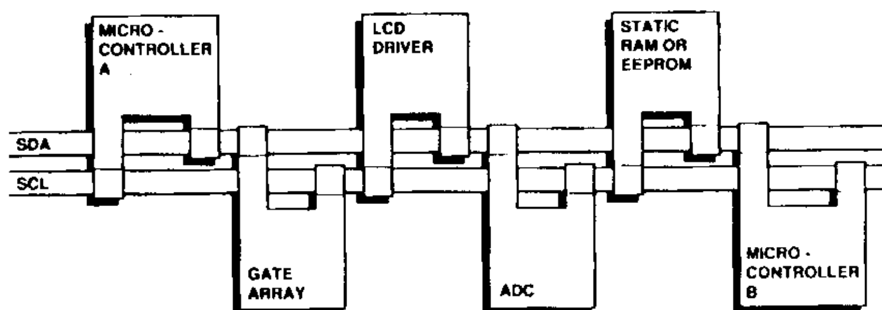
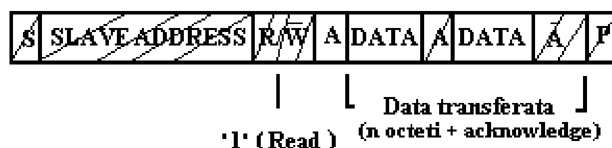


Figura 1.25. Configuratie de magistrala I^2C folosind doua microcontrollere

1. Presupunem ca microcontrollerul A doreste sa trimita informatie microcontrollerului B.
   - Microcontrollerul A (master) apeleaza (adreseaza) microcontrollerul B (slave).
   - Microcontrollerul A (emitator) transmite data microcontrollerului B (receptor).
   - Microcontrollerul A încheie transferul.

Daca bitul de directie este '1' el indica o cerere de date (READ), succesiunea de mesaje fiind (vezi figura 1.26):

2. Presupunem ca microcontrollerul A doreste sa receptioneze informatie de la microcontrollerul B.
   - Microcontrollerul A (master) se adreseaza microcontrollerului B (slave).
   - Microcontrollerul A (master-receptor) primeste data de al microcontrollerul B (slave-emitator).
   - Microcontrollerul A încheie transferul.



A = acknowledge (SDA LOW) — de la slave la master

$\overline{A}$ = not acknowledge (SDA HIGH) — de la master la slave

S = conditie de START

P = conditie de STOP

Figura 1.26. Dispozitivul Master-receptor citeste datele trimise de Slave-ul - emitator imediat dupa primul octet

Chiar si în aceasta situatie, dispozitivul master va fi A, care va genera semnalul de ceas si va încheia transferul. Transferul de date se încheie întotdeauna printr-o conditie de stop **P** generata de catre master. Totusi, daca un dispozitiv master doreste sa comunice pe magistrala, el poate genera o conditie repetata de start **Sr** si adreseaza un alt slave fara a genera întâi o conditie de stop.

1.5. MAGISTRALA ACCESS.BUS

Reprezinta magistrala de conectare a dispozitivelor accesorii la un calculator gazda, un standard introdus de catre compania Digital Equipment Corporation (actualmente înglobata în Compaq). Accesoriile sunt dispozitive periferice, de intrare / iesire, având o viteza relativ redusa fata de cea a calculatorului gazda. Ca exemple de dispozitive accesorii amintim: tastatura, scanere, cititoare de cod de bare, cititoare de cartele magnetice (card), imprimanta, convertoare de semnal, aplicatii de control în timp real etc. Topologia de conectare a dispozitivelor accesorii este de tip magistrala. Prin intermediul magistralei ACCESS pot fi conectate pâna la 125 de dispozitive periferice la un calculator gazda. Lungimea cablului de conectare poate fi pâna la 8 m. Viteza maxima de transfer pe magistrala este de 80 Kbit/s.

Magistrala ACCESS ofera avantaje atât utilizatorilor cât si dezvoltatorilor de sisteme si dispozitive periferice. Un calculator gazda necesita doar un port hardware pentru conectarea la un numar de dispozitive. Trasaturile comune în metodele de comunicare, pentru un numar mare de diverse tipuri de dispozitive, conduc la economii în dezvoltarea hardware si software.



Figura 1.27. Magistrala ACCESS.bus – o alternativa cost/performanta interfetei RS-232C

Nivelul hardware al magistralei ACCESS.bus

La nivel hardware, magistrala ACCESS se bazeaza pe principiile magistralei seriale de interconectare a circuitelor integrate (I^2C), prezentata succint anterior. Mediul fizic pentru magistrala ACCESS este compus dintr-un cablu cu patru fire izolate între ele: semnalul de date (SDA), semnalul de ceas (SCL), alimentarea (+5V) si masa (GND). Dispozitivele conectate la magistrala pot fi înlantuite prin intermediul a doi conectori. Dispozitivele portabile pot avea un cablu de conectare la magistrala principala prin intermediul unui conector în “**T**”. Semnalele seriale de ceas si date (SCL si SDA) lucreaza împreuna pentru a defini informatia transferata pe magistrala. Calculatorul gazda alimenteaza prin intermediul liniei de +5V, asigurând un curent minim de 50 mA, dispozitivele periferice. Totodata acestea pot fi alimentate si de o sursa externa.

Nivelele ierarhice ale protocolului de magistrala ACCESS.bus

Protocolul de comunicatie ACCESS.bus e compus din trei nivele: protocolul **I^2C**, protocolul de **baza** si protocolul de **aplicatie**.

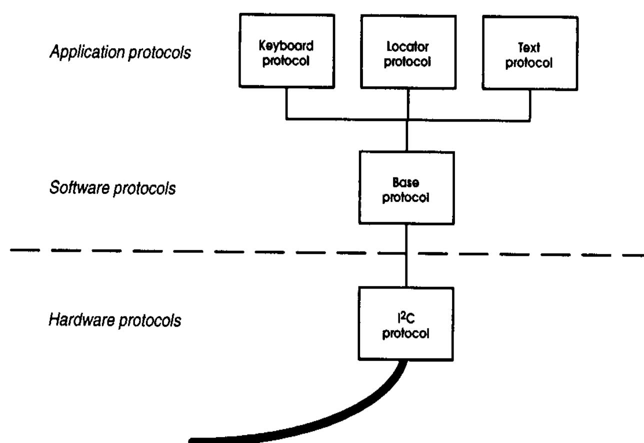


Figura 1.28. Nivelele ierarhice ale protocolului de magistrala ACCESS.bus

La nivelul cel mai de jos, apropiat de hardware, disciplina de baza a magistralei ACCESS e definita ca un subset al protocolului de magistrala I^2C. **Protocolul I^2C** defineste o magistrala simetrica de tip multimaster, în care procesul de arbitrare între dispozitivele master se efectueaza fara a pierde datele.

Nivelul de protocol urmator este **protocolul de baza**. Acest nivel e comun tuturor tipurilor de dispozitive conectate prin magistrala ACCESS si

stabileste natura asimetrica de interconectare între calculatorul gazda si un numar de dispozitive periferice. Calculatorul gazda are un rol special ca manager al magistralei. Comunicatia de date se face întotdeauna între calculator si dispozitivele periferice, niciodata între doua periferice. Daca protocolul I^2C asigura rol de conducator al unei tranzactii pe magistrala fie emitatorului fie receptorului, protocolul de comunicatie ACCESS.bus asigura rol de master exclusiv emitatorului, în timp ce rolul de slave e atribuit exclusiv receptorului. La momente diferite de timp, atât calculatorul gazda cât si dispozitivele periferice pot fi si master / emitator si slave / receptor.

Protocolul de baza al ACCESS.bus defineste formatul mesajului împachetat, transferat prin magistrala ACCESS, care reprezinta o tranzactie pe magistrala I^2C, însotita de o semantica suplimentara, incluzând sume de control. În plus, protocolul de baza defineste un set de sapte controale si tipuri de mesaje de stare care sunt folosite în procesul de configurare. Cele opt mesaje si parametrii aferenti care definesc protocolul de comunicatie ACCESS.bus sunt:

a. Mesaje de la calculatorul gazda la dispozitivele periferice:
   1. Reset ().
   2. Identificarea cererii ().
   3. Asignarea adresei (ID string, new addr) respectivului dispozitiv.
   4. Cereri de capacitate (offset) – (capabilities request) provenite de la un dispozitiv.
b. Mesaje de la dispozitivele periferice la calculatorul gazda:
   1. Atentionare (status).
   2. Identificarea raspunsului (ID string).
   3. Raspunsuri de capacitate (offset, data frag).
   4. Eroare de interfata ().

Doua caracteristici unice ale procesului de configurare sunt autoadresarea si conectarea rapida la cald. Autoadresarea se refera la modul în care dispozitivelor le sunt asignate adrese de magistrala unice în procesul de configurare fara a fi nevoie pentru a seta jumperi sau comutatori ai dispozitivelor. Conectarea rapida la cald se refera la abilitatea de atasare sau deconectare a dispozitivelor, în timp ce sistemul functioneaza fara a fi nevoie de restartarea acestuia.

Pe nivelul cel mai înalt privind protocolul de comunicatie ACCESS.bus se afla **protocolul aplicatie**. Acesta defineste semantica mesajelor specifice tipurilor functionale particulare de dispozitive. Tipuri diferite de dispozitive necesita protocoale de aplicatie diferite. Acest tip de protocol a fost definit pentru trei clase de dispozitive: **tastatura**, **dispozitive de transfer text** si **dispozitive de localizare** (locators).

Protocolul de tastatura defineste mesajele standard generate în urma apasarii tastelor si mesaje necesare controlului tastaturii. Protocolul încearca sa defineasca cel mai simplu set de functii din care poate fi construita interfata standard de tastatura.

Protocolul aferent dispozitivelor localizatoare defineste un set de mesaje standard generate în urma miscarii acestor dispozitive sau activarii unor chei întrerupatoare (comutatoare) pentru dispozitivele de pozitionare. Dispozitive mai complexe pot fi modelate ca o combinatie dintre dispozitive de baza sau pot asigura propriul lor driver.

Protocolul de comunicatie prin dispozitive cu transfer de text intentioneaza sa furnizeze un mod simplu de transmitere a datelor în format caracter sau binar, la si de la dispozitive orientate fisier, cum sunt cititoare în cod de bare sau modem-uri. Modelul de fisier secvential în format caracter serveste ca numitor comun pentru conectarea dispozitivelor interfata la RS-232C.

Un avantaj major în proiectarea dispozitivelor este acela ca ele pot împarti software-ul specific unui dispozitiv, atât la nivel firmware (rezident în dispozitiv) cât si la nivel software (rezident în driver), necesar sistemului de operare al calculatorului gazda pentru a permite programelor de aplicatie sa acceseze respectivele dispozitive. Ca si concluzie, toate cele trei nivele de protocol necesita inteligenta la nivel de dispozitiv. Nivelele de protocol joase ale acestui firmware sunt comune mai multor dispozitive. Nivelele de protocol ridicate sunt specifice functie de dispozitiv sau aplicatie.

Kit-ul de dezvoltare ACCESS.bus

ACCESS.bus este un standard industrial deschis ce asigura un mod simplu si uniform de conectare a maxim 125 de dispozitive la un singur port al unui computer. Caracteristicile principale ar fi: rata de transfer a datelor 100.000 biti / s, arbitrare hardware, reconfigurare dinamica, suporta diverse drivere de dispozitiv.

Caracteristicile kit-ului de dezvoltare software sunt:

- Satisface în întregime standardul ACCESS.bus.
- Pachetul hardware include:
  - Controller-ul de placa ACCESS.bus – 125I PC / AT.
  - Un mouse ACCESS.bus.
  - priza extensoare.
  - Cabluri ACCESS.bus (2 ft – picioare lungime si 4 ft – picioare lungime)
  - Microcontroller Philips 87C751 (în mare parte compatibil 80C51).
- Pachetul software complet contine:
  - Microcod (MC) înscris pe placa aferent controller-ului principal.

⇨ Program manager, ce functioneaza ca un program TSR (terminate and stay resident) sub DOS.
⇨ Program de monitorizare si control al magistralei ACCESS.bus.
⇨ Cod sursa pentru driver-ele software aferent atât calculatorului gazda cât si dispozitivelor periferice.
⇨ Cod sursa pentru nivelul aplicatie al protocolului ACCESS.bus.

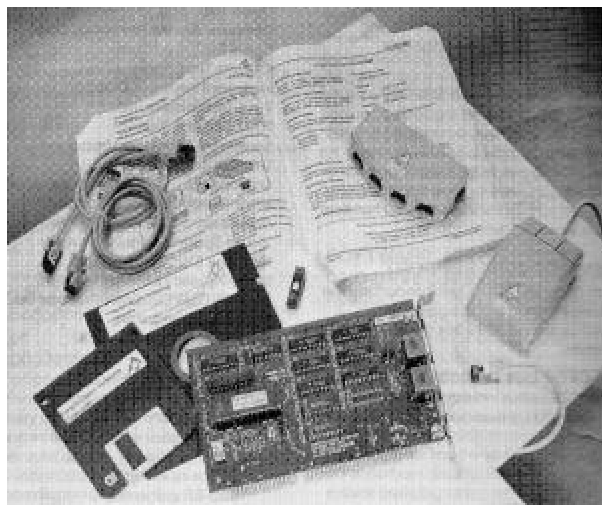


Figura 1.29. Kit-ul de dezvoltare ACCESS.bus – accesorii si software complet

Controller-ul de placa ACCESS.bus – 125I PC / AT

Se bazeaza pe microcontrollerul Philips 8xC654 cu interfata I^2C. Interfata ACCESS.bus controleaza o retea de tip ACCESS.bus. Este realizata din conectori alimentati la +5V si 0.75A. Dimensiunea retelei ACCESS.bus este de maxim 125 de dispozitive. Distanta fizica dintre dispozitive este maxim 25 ft (picioare, un picior ~ 0.3m). Interfata cu sistemul IBM PC / AT sau compatibil, se face folosind un mecanism PC / AT de intrare / iesire programabil, pe 16 biti. Adresele de I / O selectabile de utilizator sunt:

⇨ De la 0x250 la 0x25F
⇨ De la 0x260 la 0x26F
⇨ De la 0x350 la 0x35F.

Întreruperile selectabile de utilizator sunt: IRQ10, IRQ11 si IRQ12. Pe placa se afla un buffer de memorie de 8 ko SRAM (vezi figura 1.30).

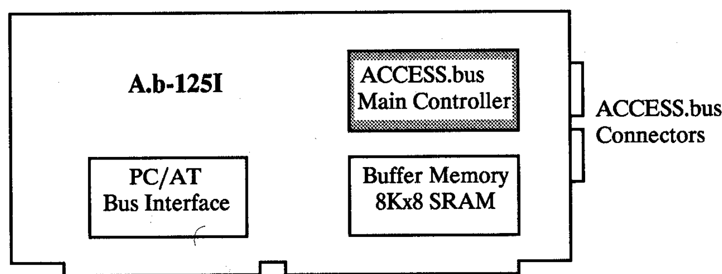


Figura 1.30. Controller-ul de placa ACCESS.bus – schema bloc

Pachetul software aferent kit-ului de dezvoltare ACCESS.bus

Microcodul (MC) înscris pe placa este un pachet de programe **în timp real** ce controleaza operatiilor diverselor dispozitive conectate la ACCESS.bus. Programul manager ruleaza ca un program rezident TSR (Terminate and Stay Resident) sub sistemul de operare DOS sau WINDOWS, comunica cu microcodul MC si cu variate drivere de dispozitiv. El ruteaza mesajele de aplicatie si control între dispozitivele fizice si driverele lor software. Programul monitor, este bazat pe meniuri, usor accesibile utilizatorului, afiseaza mesajele selectate de utilizator si permite acestuia sa controleze dispozitivele specifice. La alimentarea placii este realizat un test complet de diagnosticare proprie (memorie, periferice aferente). Diagnosticarea se realizeaza sub controlul programului monitor.

Protocolul CAN (retea de control)

CAN este un protocol de multiplexare al instalatiilor electrice dezvoltat de firma Bosch pentru aplicatii industriale automatizate, masini si utilaje, echipamene medicale, echipamente de control în constructii. Protocolul este atractiv pentru utilizarea într-o varietate de aplicatii deoarece reprezinta un instrument puternic de detectie a erorilor. Poate fi utilizat cu succes în medii cu nivel de zgomot ridicat sau critic. CAN este foarte flexibil în termenii transmisiei de date si schemei de conectare si poate fi usor adaptat la majoritatea aplicatiilor.

Compania Philips ofera o varietate de dispozitive care suporta protocolul CAN, cum ar fi: dispozitive “stand-alone” (de sine - statatoare) dar si microcontrollere cu interfata CAN integrata. Dintre acestea amintim: 82C200 – controller stand alone, 82C150 (dispozitive periferice legate serial la CAN) si 82C250 (controller de emisie – receptie legat la CAN). Exemple de microcontrollere care au integrate o interfata CAN sunt 8xC592 si 8x598.

1.6. PLACA DE DEZVOLTARE DB – 51

DB-51 este o placa de dezvoltare / proiectare a unui sistem de înalta performanta dedicat familiei de microcontollere Philips 80C51. DB-51 reprezinta un instrument flexibil, usor de folosit care permite utilizatorului sa construiasca un prototip primar, sa-l analizeze si sa-l depaneze, sa faca schimbari asupra sa si sa continue depanarea. Îmbunatatirea deciziilor de proiectare se face folosind DB-51 pentru a verifica si testa avantajele câtorva microcontrollere diferite. De asemenea, placa de dezvoltare DB-51 reprezinta un instrument ideal de antrenare pentru familiarizarea utilizatorului cu proiectarea, folosind arhitectura 80C51. De remarcat ca, DB-51 nu intentioneaza totusi sa înlocuiasca un sistem de emulare complet în proiectarea complexa cu microcontrollere.

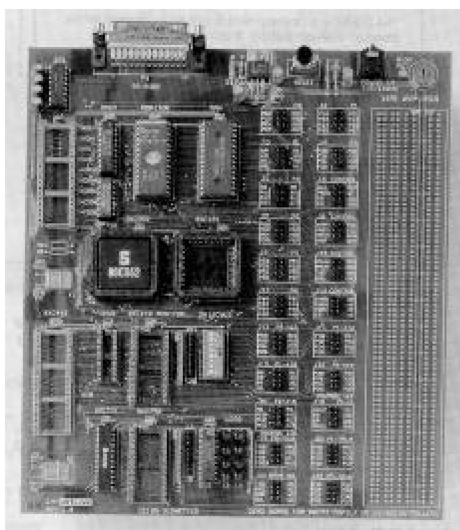


Figura 1.31. Placa de dezvoltare DB-51

Caracteristici de baza

- Suporta majoritatea microcontrollerelor derivate ale familiei Philips 80C51 (8x31/51, 8x32/52, 8xC31/51, 8xC32/52, 8xC652, 8xC654, 8xC851, 8xC550, 8xC552, 8xC562, 8xC451, 8xC528 si altele cu memorie externa adresabila si suport UART).
- Se conecteaza prin serial la un calculator IBM – PC sau la alte calculatoare gazda compatibile.

- Sistemul de memorie al DB-51 consta din 32 ko RAM. În acest spatiu se încarca si modifica programele utilizator.
- Contine puncte de întrerupere software (breakpoint). Acestea permit executia programelor în timp real pâna când opcode-ul instructiunii, pe care s-a setat punctul de întrerupere, este citit de la respectiva adresa.
- Examineaza si altereaza continutul registrilor, memoriei RAM si porturile.
- Contine un debugger simbolic, compatibil cu link-editorul de fisiere obiect. DB-51 permite depanarea simbolica pas cu pas a programelor, atât în limbaj de asamblare cât si în limbaje de nivel înalt (PLM, C). Debugger-ul foloseste simboluri continute în fisiere absolute (complete), generate de majoritatea programelor de relocare si link-editare. Debuggerul obisnuieste sa încarce un program, sa-l execute în timp real sau sa simuleze mediul software, sa programeze microcontrollerul sau sa execute multe alte functii. Programele pot fi executate în mod continuu (**run**) sau pas cu pas (**trace**). Depanarea poate fi facuta folosind linii de program scrise în limbaj de nivel înalt sau instructiuni în asamblare. Debuggerul trebuie sa permita cunoasterea permanenta (prin intermediul unei ferestre) a starii programului din diverse perspective, cum ar fi: a variabilelor si a valorilor lor, a punctelor de întrerupere, a fisierului sursa, a registrelor procesorului, a locatiilor memoriei, a registrelor periferice. Variabilele programului pot fi inspectate, iar valorile aferente lor pot fi retinute pe toata durata rularii programului. De asemenea, valoarea curenta a unei variabile poate fi înlocuita cu una specificata. Pentru depanarea simbolica a programelor sursa, trebuie ca acestea sa fie prevazute (pregatite) cu informatia de depanare (numarul fiecarei linii de cod, referinte globale si locale, etichete etc). Pregatirea unui program pentru depanare se realizeaza în etapele:
  1. *Scrierea* codului sursa cu ajutorul unui **Editor**.
  2. *Compilarea* surselor de catre un **Asamblor** sau un Cross-**compilator** de limbaje de nivel înalt.
  3. *Localizarea* si *link-editarea* fisierelor obiect cu programul **Intel RL51** sau cu unul asemanator, program care genereaza un format compatibil cu cel al procesorului Intel.
- Reprezinta un analizor de performanta.
- Încarca si descarca fisiere în format ASCII si obiect.
- Se furnizeaza împreuna cu un ghid de utilizare, dotat cu exemple si aplicatii destinate familiarizarii utilizatorului cu arhitectura 8xC51, programarea si utilizarea placii de dezvoltare / proiectare DB-51.
- Limitari:

- ⇨ Programul monitor foloseste partea inferioara a celor 32 ko de memorie.
- ⇨ Standardul UART (Universal Asynchron Receiver / Transmiter) este folosit la comunicatia cu PC-ul si astfel, nu este în mod normal disponibil programului utilizator.
- ⇨ Raspunsul la întreruperi este întârziat usor prin redirectarea de la programul monitor la programul uilizator.
- ⇨ Folosirea circuitelor numaratoare de tip "**watchdog**" sau "**power-down**" pentru avertizarea împotriva caderii sursei de alimentare, sau modurile de operare lente, temporizatoare sunt limitate datorita interactiunii cu programul monitor.

- ➢ Software-ul utilizator furnizat de placa DB-51 este caracterizat atât de un program bazat pe meniuri cât si de o interfata software bazata pe linia de comanda. Asamblorul si dezasamblorul sunt furnizate împreuna, cu posibilitati de încarcare si descarcare a fisierelor în format ascii sau obiect.
- ➢ Setul de comenzi acceptate:
  ASM – BIT – BYTE – BREAKPOINT [enable, disable, reset] – CHIP [type] – CLS – CODE – DATA – DASM – DEFAULT – DIR – EVALUATE – EXIT – GO [from, till] – HALT – HELP – HISTORY – LINES – LIST [file] – LOAD [code, symbols] – LOCALS – MODULES – PORTS – PROCEDURES – PUBLICS – RBIT – RBYTE – REGISTERS – RESET – SAVE – SOUND – STATUS – STEP [n] - TIME
- ➢ Caracteristicile calculatorului gazda:
  IBM PC / AT / XT sau compatibile în configuratia 512 ko RAM minim, un disc floppy, o interfata RS – 232 pentru PC, cablu, sistem de operare PC DOS versiunea minim 6.0 .

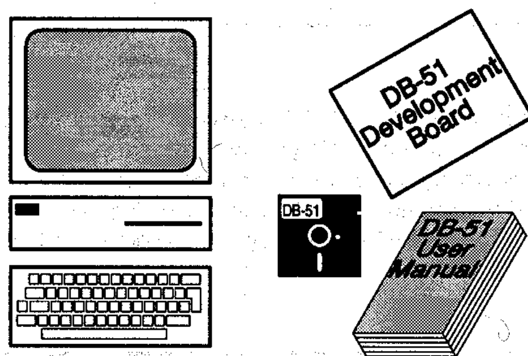


Figura 1.32. Proiectarea folosind placa DB-51

2. ARHITECTURA MICROPROCESOARELOR ACTUALE

2.1. MODELUL DE MICROPROCESOR SCALAR RISC

Microprocesoarele RISC (Reduced Instruction Set Computer) au aparut ca o replica la lipsa de eficienta a modelului conventional de procesor de tip CISC (Complex Instruction Set Computer – inamicii conceptului RISC spuneau Complete Instruction Set Computer). Multe dintre instructiunile masina ale procesoarelor CISC sunt folosite rar în softul de baza, cel care implementeaza sistemele de operare, utilitarele, translatoarele, etc. Lipsa de eficienta a modelului conventional CISC a fost pusa în evidenta prin anii '80 de arhitecturi precum INTEL 80x86, MOTOROLA 680x0, iar in domeniul sistemelor în special de catre arhitecturile VAX-11/780 si IBM - 360,370, cele mai cunoscute la acea vreme. Modelele CISC sunt caracterizate de un set foarte bogat de instructiuni - masina, formate de instructiuni de lungime variabila, numeroase moduri de adresare deosebit de sofisticate, lipsa unei interfete hardware – software optimizate etc. Evident ca aceasta complexitate arhitecturala are o repercursiune negativa asupra performantei masinii.

Din punct de vedere istoric, primele microprocesoare RISC s-au proiectat la IBM si respectiv la Universitatea Berkeley, USA (1981). Spre deosebire de CISC-uri, proiectarea sistemelor RISC are în vedere ca înalta performanta a procesarii se poate baza pe simplitatea si eficacitatea proiectului. Strategia de proiectare a unui microprocesor RISC trebuie sa tina cont de analiza aplicatiilor posibile pentru a determina operatiile cele mai frecvent utilizate, precum si optimizarea structurii hardware în vederea unei executii cât mai rapide a instructiunilor. Dintre aplicatiile specifice sistemelor RISC se amintesc: conducerea de procese în timp real, procesare de semnale (DSP – Digital Signal Processing), calcule stiintifice cu viteza ridicata,

grafica de mare performanta, elemente de procesare în sisteme multiprocesor si alte sisteme cu prelucrare paralela etc.

Caracteristicile de baza ale modelului RISC sunt urmatoarele:

- Timp de proiectare si erori de constructie mai reduse decât la variantele CISC comparabile.
- Unitate de comanda hardware în general cablata, cu firmware redus sau deloc, ceea ce mareste rata de executie a instructiunilor.
- Utilizarea tehnicilor de procesare pipeline a instructiunilor, ceea ce implica o rata teoretica de executie de o instructiune / ciclu, pe modelele de procesoare care pot lansa în executie la un moment dat o singura instructiune (procesoare scalare).
- Memorie sistem de înalta performanta, prin implementarea unor arhitecturi avansate de memorie cache si MMU (Memory Management Unit). De asemenea contin mecanisme hardware de memorie virtuala bazate în principal pe paginare, ca si sistemele CISC de altfel.
- Set relativ redus de instructiuni simple, majoritatea fara referire la memorie si cu putine moduri de adresare. În general, doar instructiunile LOAD / STORE sunt cu referire la memorie (arhitectura tip LOAD / STORE). La implementarile recente caracteristica de "set redus de instructiuni" nu trebuie înteleasa add literam ci mai corect în sensul de set optimizat de instructiuni în vederea implementarii aplicatiilor propuse (în special implementarii limbajelor de nivel înalt - C, C++, Pascal, etc.).
- Datorita unor particularitati ale procesarii pipeline (în special hazardurile pe care aceasta le implica), apare necesitatea unor compilatoare optimizate zise si reorganizatoare sau schedulere, cu rolul de a reorganiza programul sursa pentru a putea fi procesat optimal din punct de vedere al timpului de executie.
- Format fix al instructiunilor, codificate în general pe un singur cuvânt de 32 biti, mai recent pe 64 biti (Alpha 21264, IA-64 Merced etc.).
- Necesitati de memorare a programelor mai mari decât în cazul microsistemelor conventionale, datorita simplitatii instructiunilor cât si reorganizatoarelor care pot actiona defavorabil asupra "lungimii" programului obiect.
- Set de registre generale substantial mai mare decât la CISC-uri, în vederea lucrului "în ferestre" (register windows), util în optimizarea instructiunilor CALL / RET. Numarul mare de registre generale este util si pentru marirea spatiului intern de procesare, tratarii optimizate a evenimentelor de exceptie, modelului ortogonal de programare, etc.

Registrul R0 este cablat la zero în majoritatea implementarilor, pentru optimizarea modurilor de adresare si a instructiunilor [3].

În proiectarea setului de instructiuni aferent unui microprocesor RISC intervin o multitudine de consideratii dintre care se amintesc:

a) Compatibilitatea cu seturile de instructiuni ale altor procesoare pe care s-au dezvoltat produse software consacrate (compatibilitatea de “sus în jos”, valabila de altfel si la CISC-uri). Portabilitatea acestor produse pe noile procesoare este conditionata de aceasta cerinta care vine în general în contradictie cu cerintele de performanta ale sistemului.
b) În cazul microprocesoarelor, setul de instructiuni este în strânsa dependenta cu tehnologia folosita, care de obicei limiteaza sever performantele (constrângeri legate de aria de integrare, numarul de pini, cerinte restrictive particulare ale tehnologiei, etc.).
c) Minimizarea complexitatii unitatii de comanda precum si minimizarea fluxului de informatie procesor - memorie.
d) În cazul multor microprocesoare RISC setul de instructiuni masina este ales ca suport pentru implementarea unor limbaje de nivel înalt.

Setul de instructiuni al unui microprocesor RISC este caracterizat de simplitatea formatului precum si de un numar limitat de moduri de adresare. De asemenea, se urmareste ortogonalizarea setului de instructiuni.

Conceptul cel mai important însa în cazul acestor microprocesoare consta în implementarea tehnicii pipeline de procesare a instructiunilor. Ideea nu era noua dar arhitectura RISC definita succint anterior se mapeaza optimal pe acest concept. Tehnica de procesare pipeline reprezinta o tehnica de procesare paralela a informatiei prin care un proces secvential este divizat în subprocese, fiecare subproces fiind executat într-un segment special dedicat si care opereaza în paralel cu celelalte segmente. Fiecare segment executa o procesare partiala a informatiei. Rezultatul obtinut în segmentul i este transmis în tactul urmator spre procesare segmentului (i+1). Rezultatul final este obtinut numai dupa ce informatia a parcurs toate segmentele, la iesirea ultimului segment. Denumirea de pipeline provine de la analogia cu o banda industriala de asamblare. Este caracteristic acestor tehnici faptul ca diversele procese se pot afla în diferite faze de prelucrare în cadrul diverselor segmente, simultan. Suprapunerea procesarilor este posibila prin asocierea unui registru de încarcare fiecarui segment din pipeline. Registrele produc o separare între segmente astfel încât fiecare segment sa poata prelucra date separate (vezi Figura 2.1). În figura 2.1. s-au notat prin Ti - registrii tampon iar prin Ni - nivelele de prelucrare (combinationale sau secventiale).

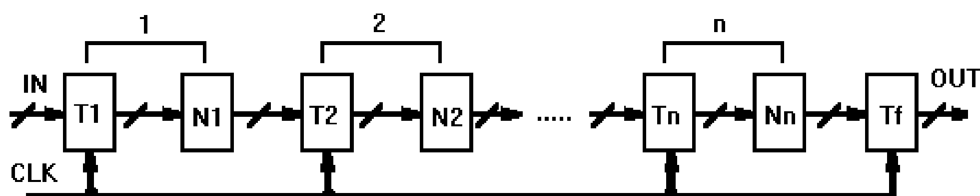


Figura 2.1. Structura de procesare tip pipeline.

Procesarea pipeline a instructiunilor reprezinta o tehnica de procesare prin intermediul careia fazele (ciclii) aferente multiplelor instructiuni sunt suprapuse în timp. Se întelege printr-o faza aferenta unei instructiuni masina o prelucrare atomica a informatiei care se desfasoara dupa un algoritm implementat în hardware (firmware) si care dureaza unul sau mai multi tacti. În acest sens se exemplifica: faza de aducere (fetch) a instructiunii, faza de decodificare, faza de executie, faza de citire / scriere data, etc. Arhitectura microprocesoarelor RISC este mai bine adaptata la procesarea pipeline decât cea a sistemelor conventionale CISC, datorita instructiunilor de lungime fixa, a modurilor de adresare specifice, a structurii interne bazate pe registre generale, etc. Microprocesoarele RISC uzuale detin o structura pipeline de instructiuni întregi pe 4 - 6 nivele. De exemplu, microprocesoarele companiei MIPS au urmatoarele 5 nivele tipice:

1. Nivelul **IF (instruction fetch)** - se calculeaza adresa instructiunii ce trebuie citita din cache-ul de instructiuni sau din memoria principala si se aduce instructiunea;
2. Nivelul **RD (ID)** - se decodifica instructiunea adusa si se citesc operanzii din setul de registri generali. În cazul instructiunilor de salt, pe parcursul acestei faze se calculeaza adresa de salt;
3. Nivelul **ALU** - se executa operatia ALU asupra operanzilor selectati în cazul instructiunilor aritmetico-logice; se calculeaza adresa de acces la memoria de date pentru instructiunile LOAD / STORE;
4. Nivelul **MEM** - se acceseaza memoria cache de date sau memoria principala, însa numai pentru instructiunile LOAD / STORE. Acest nivel pe functia de citire poate pune probleme datorate neconcordantei între rata de procesare si timpul de acces la memoria principala. Rezulta deci ca într-o structura pipeline cu N nivele, memoria trebuie sa fie în principiu de N ori mai rapida decât într-o structura de calcul conventionala. Acest lucru se realizeaza prin implementarea de arhitecturi de memorie rapide (cache, memorii cu acces întretesut, etc.). Desigur ca un ciclu cu MISS în cache pe acest nivel (ca si pe nivelul IF de altfel), va determina stagnarea temporara a acceselor la memorie sau chiar a

procesarii interne. La scriere, problema aceasta nu se pune datorita procesorului de iesire specializat DWB care lucreaza în paralel cu procesorul central dupa cum deja am aratat.

5. Nivelul **WB (write buffer)** - se scrie rezultatul ALU sau data citita din memorie (în cazul unei instructiuni LOAD) în registrul destinatie din setul de registri generali ai microprocesorului.

Prin urmare, printr-o astfel de procesare se urmareste o rata ideala de o instructiune / ciclu masina ca în Figura 2.2, desi dupa cum se observa, timpul de executie pentru o instructiune data nu se reduce.

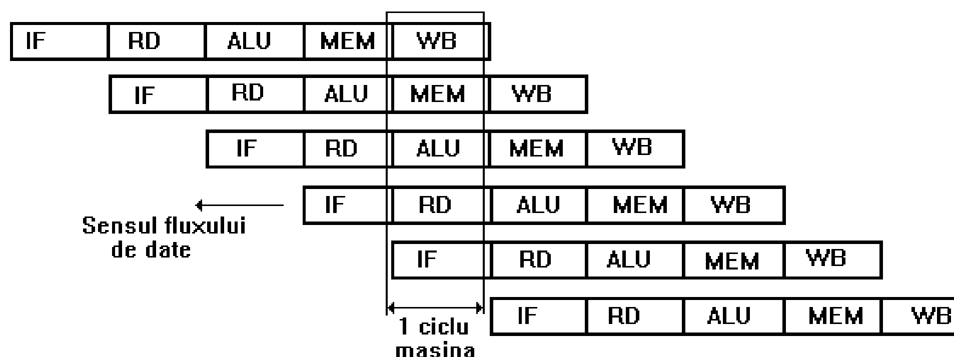


Figura 2.2. Principiul procesarii pipeline într-un procesor RISC

Se observa imediat necesitatea suprapunerii a 2 nivele concurentiale: nivelul IF si respectiv nivelul MEM, ambele cu referire la memorie. În cazul microprocesoarelor RISC aceasta situatie se rezolva deseori prin legaturi (busuri) separate între procesor si memoria de date respectiv de instructiuni (arhitectura Harvard).

În literatura se citeaza un model de procesor numit superpipeline. Acesta este caracterizat printr-un numar relativ mare al nivelelor de procesare. Desigur ca în acest caz detectia si corectia hazardurilor de date si ramificatii este mai dificila în acest caz. Arhitecturile superpipeline se preteaza la tehnologiile cu grade de împachetare reduse unde nu este posibila multiplicarea resurselor hardware, în schimb caracterizate prin viteze de comutatie ridicate (ECL, GaAs). O asemenea arhitectura caracterizeaza de ex. procesoarele din familia DEC (Digital Equipment Corporation, azi înglobata de catre Compaq) Alpha. Avantajul principal al arhitecturilor superpipeline este ca permit frecvente de tact deosebit de ridicate (600-1000 MHz la nivelul tehnologiilor actuale), aspect normal având în vedere super - divizarea stagiilor de procesare. Fluxul procesarii pipeline poate fi stagnat de anumite evenimente nedorite numite hazarduri.

Acestea se împart principial în 3 categorii distincte: structurale, de date si de ramificatie.

Hazardurile structurale sunt determinate de conflictele la resurse comune, adica atunci când mai multe procese simultane aferente mai multor instructiuni în curs de procesare, acceseaza o resursa comuna. Pentru a le elimina prin hardware, se impune de obicei multiplicarea acestor resurse. De exemplu, un procesor care are un set de registri generali de tip uniport si în anumite situatii exista posibilitatea ca 2 procese sa doreasca sa scrie în acest set simultan. O alta situatie de acest fel, dupa cum deja am mai aratat, poate consta în accesul simultan la memorie a 2 procese distincte: unul de aducere a instructiunii (IF), iar celalalt de aducere a operandului sau scriere a rezultatului în cazul unei instructiuni LOAD / STORE (nivelul MEM). Dupa cum am mai aratat, aceasta situatie se rezolva în general printr-o arhitectura Harvard a busurilor si cache-urilor.

Hazardurile de date apar când o instructiune depinde de rezultatele unei instructiuni anterioare în banda. Pot fi la rândul lor clasificate în 3 categorii (RAW, WAR, WAW), dependent de ordinea acceselor de citire respectiv scriere, în cadrul instructiunilor.

Considerând instructiunile i si j succesive, hazardul RAW (Read After Write) apare atunci când instructiunea j încearca sa citeasca o sursa înainte ca instructiunea i sa scrie în aceasta. Apare deosebit de frecvent în implementarile actuale de procesoare pipeline si este singura stagnare obiectiva, care implica secventialitatea executiei instructiunilor respective.

Hazardul WAR (Write After Read) poate sa apara atunci când instructiunea j scrie o destinatie înainte ca aceasta sa fie citita pe post de sursa de catre o instructiune anterioara notata i. Poate sa apara când într-o structura pipeline exista o faza de citire posterioara unei faze de scriere. De exemplu, modurile de adresare indirecta cu predecrementare pot introduce acest hazard, de aceea ele nici nu sunt implementate în arhitecturile de tip RISC. De precizat ca aceste hazarduri WAR, pot apare si datorita executiei instructiunilor în afara ordinii lor normale, din program (executie Out of Order). Aceasta procesare Out of Order este impusa de cresterea performantei si se poate realiza atât prin mijloace hardware cât si software, legat de optimizarea programelor pe arhitecturile pipeline.

Hazardul WAW (Write After Write), apare atunci când instructiunea j scrie un operand înainte ca acesta sa fie scris de catre instructiunea i. Asadar, în acest caz scrierile s-ar face într-o ordine eronata. Hazardul WAW poate aparea în structurile care au mai multe nivele de scriere sau care permit unei instructiuni sa fie procesata chiar daca o instructiune anterioara este blocata. Modurile de adresare indirecta cu postincrementare pot introduce acest hazard, fapt pentru care ele sunt evitate în procesoarele

RISC. De asemenea, acest hazard poate sa apara in cazul executiei Out of Order a instructiunilor care au aceeasi destinatie. Hazardurile WAR si WAW sunt doar conflicte de nume si prin redenumirea resurselor implicate, ele dispar. Astfel, acestea nu mai implica în mod necesar o secventialitate In Order a executiei instructiunilor respective [3].

Hazardurile de ramificatie pot fi generate de catre instructiunile de ramificatie (branch). Cauzeaza pierderi de perfomanta în general mai importante decât hazardurile structurale si de date, mai ales la procesoarele superscalare. Efectele defavorabile ale instructiunilor de ramificatie pot fi reduse prin metode soft (reorganizarea programului sursa), sau prin metode hard care determina în avans daca saltul se va face sau nu (branch prediction) si calculeaza în avans noul PC (program counter). Diverse statistici arata ca instructiunile de salt neconditionat au o frecventa între 2 - 8% din instructiunile unui program de uz general, iar cele de salt conditionat între 11 - 17%. S-a aratat ca salturile conditionate simple se fac cu o probabilitate de cca. 50%, loop-urile cu o probabilitate de cca. 90%, iar majoritatea salturilor orientate pe bit nu se fac.

Acest model de procesare are consecinte imediate extrem de criticabile precum:

- Instructiunile de pe calea "*cea rea*" cauzeaza hazarduri structurale si deci încetiniri ale procesarii.
- Aceleasi instructiuni pot conduce la miss-uri în cacheuri având consecinte defavorabile asupra procesului de fetch al caii "*corecte*".
- Speculatiile "*în adâncime*" amplifica exponential dezavantajele mai sus mentionate si de asemenea "*foamea*" de resurse hardware.

2.2. MICROARHITECTURI CU EXECUTII MULTIPLE ALE INSTRUCTIUNILOR

Microprocesoarele avansate actuale ating, teoretic cel putin, rate medii de procesare de mai multe instructiuni per tact. Procesoarele care initiaza executia mai multor operatii simultan intr-un ciclu (sau tact) se numesc procesoare cu executii multiple ale instructiunilor. Un astfel de procesor aduce din cache-ul de instructiuni una sau mai multe instructiuni simultan si le distribuie spre executie în mod dinamic sau static (prin reorganizatorul de program), multiplelor unitati de executie.

Principiul acestor procesoare paralele numite si "masini cu executie multipla" (MEM) consta în existenta mai multor unitati de executie paralele, care pot avea latente diferite. Pentru a facilita procesarea acestor instructiuni, acestea sunt codificate pe un singur cuvânt de 32 sau 64 de biti uzual, pe modelul RISC anterior prezentat. Exista desigur si implementari MEM realizate pe modele de programare de tip CISC, cum este cazul lui Intel Pentium, dar si aici principiile RISC s-au impus, cel putin la nivel de microprogram. Daca decodificarea instructiunilor, detectia dependentelor de date dintre ele, rutarea si lansarea lor în executie din bufferul de prefetch înspre unitatile functionale se fac prin hardware, aceste procesoare MEM se mai numesc si superscalare. Pot exista mai multe unitati functionale distincte, dedicate de exemplu diverselor tipuri de instructiuni tip întreg sau flotant. În cazul procesoarelor MEM, paralelismul temporal determinat de procesarea pipeline se suprapune cu un paralelism spatial determinat de existenta mai multor unitati de executie. În general structura pipeline a coprocesorului are mai multe nivele decât structura pipeline a procesorului ceea ce implica probleme de sincronizare mai dificile decât în cazul procesoarelor pipeline scalare. Acelasi lucru este valabil si între diferite alte tipuri de instructiuni având latente de executie diferite. Caracteristic deci procesoarelor superscalare este faptul ca dependentele de date între instructiuni se rezolva prin hardware, în momentul decodificarii instructiunilor. Modelul intuitiv ideal de procesare superscalara, în cazul unui procesor care poate aduce si decodifica 2 instructiuni simultan este prezentat în Figura 2.3.

Este evident ca în cazul superscalar complexitatea logicii de control este mult mai ridicata decât în cazul pipeline scalar, întrucât detectia si sincronizarile între structurile pipeline de executie cu latente diferite si care lucreaza în paralel devin mult mai dificile. De exemplu un procesor superscalar având posibilitatea aducerii si executiei a "N" instructiuni masina simultan, necesita N(N-1) unitati de detectie a hazardurilor de date între aceste instructiuni (comparatoare digitale), ceea ce conduce la o complexitate ridicata a logicii de control.

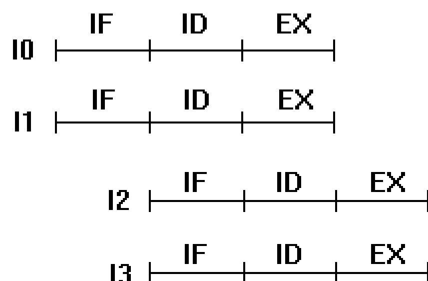


Figura 2.3. Modelul executiei superscalare

Procesoarele VLIW (Very Long Instruction Word) reprezinta procesoare care se bazeaza pe aducerea în cadrul unei instructiuni multiple a mai multor instructiuni RISC independente pe care le distribuie spre procesare unitatilor de executie. Asadar, rata de executie ideala la acest model, este de n instructiuni/ciclu. Pentru a face acest model viabil, sunt necesare instrumente soft de exploatare a paralelismului programului, bazate pe gruparea instructiunilor simple independente si deci executabile în paralel, în instructiuni multiple. Arhitecturile VLIW sunt tot de tip MEM. Principiul VLIW este sugerat intuitiv în Figura 2.4:

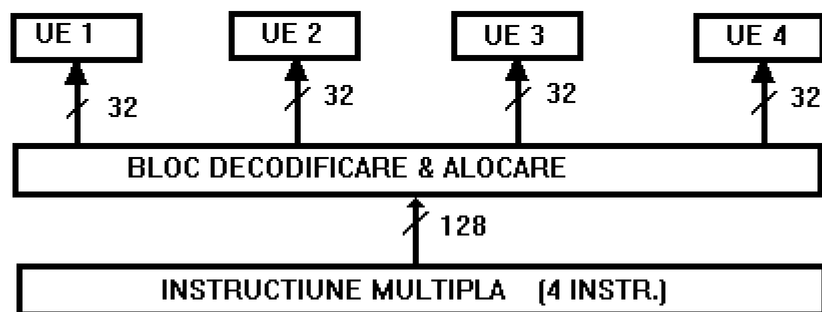


Figura 2.4. Decodificarea si alocarea instructiunilor într-un procesor VLIW

În cadrul acestui model, se încearca prin transformari ale programului, ca instructiunile RISC primitive din cadrul unei instructiuni multiple sa fie independente si deci sa se evite hazardurile de date între ele, a caror gestionare ar fi deosebit de dificila în acest caz. Performanta procesoarelor VLIW este esential determinata de programele de compilare si reorganizare care trebuie sa fie deosebit de "inteligente". De aceea acest model de arhitectura se mai numeste uneori si EPIC (Explicitly Parallel Instruction

Computing – terminologie Intel Co. folosita în documentatia microprocesorului IA-64 Merced).

Prin urmare, în cazul modelului de procesor VLIW, compilatorul trebuie sa înglobeze mai multe instructiuni RISC primitive independente în cadrul unei instructiuni multiple, în timp ce în cazul modelului superscalar, rezolvarea dependentelor între instructiuni se face prin hardware, începând cu momentul decodificarii acestor instructiuni. De asemenea, pozitia instructiunilor primitive într-o instructiune multipla determina alocarea acestor instructiuni primitive la unitatile de executie, spre deosebire de modelul superscalar unde alocarea se face dinamic prin control hardware. Acest model de procesor nu mai necesita sincronizari si comunicatii de date suplimentare între instructiunile primitive dupa momentul decodificarii lor, fiind astfel mai simplu din punct de vedere hardware decât modelul superscalar. Un model sugestiv al principiului de procesare VLIW este prezentat în Figura 2.5.

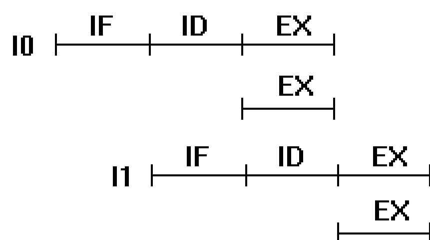


Figura 2.5. Principiul de procesare VLIW

Având în vedere ideile de implementare a executiilor multiple, o arhitectura superscalara reprezentativa este prezentata în Figura 2.6. Mentionam ca la ora actuala, pe plan comercial, piata este dominata de microprocesoarele superscalare, în principal datorita dezideratelor de compatibilitate între versiunile din cadrul aceleiasi familii. Prin SR am notat statiile de rezervare aferente unitatilor de executie ale procesorului. Acestea implementeaza printre altele bufferul "instruction window" necesar procesoarelor superscalare cu executie Out of Order. Numarul optim de locatii al fiecarei SR se determina pe baza de simulare.

Desi performanta maxima a unei asemenea arhitecturi ar fi de 6 instructiuni/ciclu, în realitate, bazat pe simulari ample, s-a stabilit ca rata medie de executie este situata între 1-2 instructiuni / ciclu. În sub 1% din cazuri, masurat pe benchmark-uri nenumerice, exista un potential de paralelism mai mare de 6 instructiuni / ciclu în cazul unei arhitecturi superscalare "pure". Aceasta se datoreaza în primul rând capacitatii limitate

a bufferului de prefetch care constituie o limitare principiala a oricarui procesor, exploatarea paralelismului între instructiuni fiind limitata de capacitatea acestui buffer. În tehnologia actuala acesta poate memora între 8 - 64 instructiuni, capacitati mai mari ale acestuia complicând mult logica de detectie a hazardurilor RAW dupa cum am aratat. În continuare se prezinta pe scurt rolul modulelor componente din aceasta schema tipica.

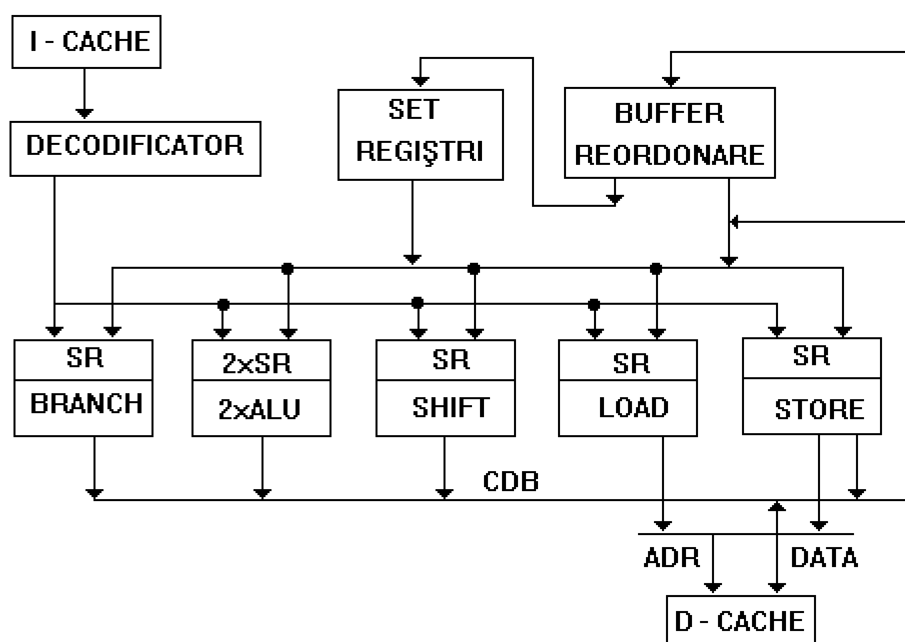


Figura 2.6. Arhitectura tipica a unui procesor superscalar

Decodificatorul plaseaza instructiunile multiple în SR - urile corespunzatoare. O unitate functionala poate starta executia unei instructiuni din SR imediat dupa decodificare daca instructiunea nu implica dependente, operanzii îi sunt diponibili si daca unitatea de executie este libera. În caz contrar, instructiunea asteapta în SR pâna când aceste conditii vor fi îndeplinite. Daca mai multe instructiuni dintr-o SR sunt simultan disponibile spre a fi executate, procesorul o va selecta pe prima din secventa de instructiuni. Desigur ca este necesar un mecanism de arbitrare în vederea accesarii busului de date comun (CDB – Common Data Bus) de catre diversele unitati de executie (UE). În vederea cresterii eficientei, deseori magistralele interne sunt multiplicate.

Bufferul de reordonare (RB - Reorder Buffer) este în legatura cu mecanismul de redenumire dinamica a registrilor în vederea executiei Out of Order precum si cu necesitatea implementarii unui mecanism precis de

tratare a evenimentelor de exceptie (derute, devieri, întreruperi hard-soft, etc.). Acest deziderat nu este deloc trivial având în vedere procesarea pipeline a instructiunilor si posibilitatea aparitiei unor exceptii imprecise. Acest RB contine un numar de locatii care sunt alocate în mod dinamic rezultatelor instructiunilor.

În urma decodificarii unei instructiuni, rezultatul acesteia este asignat unei locatii din RB, iar numarul registrului destinatie este asociat acestei locatii. În acest mod, registrul destinatie este practic redenumit printr-o locatie din RB. În urma decodificarii se creaza prin hard un "tag" care reprezinta numele unitatii de executie care va procesa rezultatul instructiunii respective. Acest tag va fi scris în aceeasi locatie din RB. Din acest moment, când o instructiune urmatoare face referire la respectivul registru pe post de operand sursa, ea va apela în locul acestuia valoarea înscrisa în RB sau, daca valoarea nu a fost înca procesata, tag-ul aferent locatiei. Daca mai multe locatii din RB contin acelasi numar de registru (mai multe instructiuni în curs au avut acelasi registru destinatie), se va genera locatia cea mai recent înscrisa (tag sau valoare).

Este evident deja ca RB se implementeaza sub forma unei memorii asociative, cautarea facându-se dupa numarul registrului destinatie la scriere, respectiv sursa la citire. Daca accesarea RB se soldeaza cu miss, atunci operandul sursa va fi citit din setul de registri. În caz de hit, valoarea sau tag-ul citite din RB sunt memorate în SR corespunzatoare. Când o unitate de executie genereaza un rezultat, acesta se va înscrie în SR si în locatia din RB care au tag-ul identic cu cel emis de catre respectiva unitate. Rezultatul înscris într-o SR poate debloca anumite instructiuni aflate în asteptare. Dupa ce rezultatul a fost scris în RB, instructiunile urmatoare vor continua sa-l citeasca din RB ca operand sursa pâna când va fi evacuat si scris în setul de registri. Evacuarea se va face în ordinea secventei originale de instructiuni pentru a se putea evita exceptiile imprecise. Asadar, redenumirea unui registru cu o locatie din RB se termina în momentul evacuarii acestei locatii.

Bufferul RB poate fi gestionat ca o memorie FIFO (First In First Out). În momentul decodificarii unei instructiuni, rezultatul acesteia este alocat în coada RB. Rezultatul instructiunii este înscris în momentul în care unitatea de executie corespunzatoare îl genereaza. Când acest rezultat ajunge în prima pozitie a RB, daca între timp nu au aparut exceptii, este înscris în setul de registri. Daca instructiunea nu s-a încheiat atunci când locatia alocata în RB a ajuns prima, bufferul RB nu va mai avansa pâna când aceasta instructiune nu se va încheia. Decodificarea instructiunilor poate însa continua atât timp cât mai exista locatii disponibile în RB.

2.3. OPTIMIZAREA PROGRAMELOR OBIECT. TEHNICI MODERNE DE PROCESARE

Ca si în cazul procesoarelor scalare, reorganizarea programelor (scheduling) reprezinta procesul de aranjare a instructiunilor din cadrul unui program obiect astfel încât acesta sa se execute pe arhitectura hardware într-un mod cvasioptimal din punct de vedere al timpului de procesare. Procesul de reorganizare a instructiunilor determina cresterea probabilitatii ca procesorul sa aduca simultan din cache-ul de instructiuni mai multe instructiuni independente. De asemenea asigura procesarea eficienta a operatiilor critice din punct de vedere temporal în sensul reducerii prin masacare a latentelor specifice acestor operatii. Este rezolvata demult problema optimizarii "basic block"-urilor, adica a acelor unitati secventiale de program care nu contin ramificatii si nu sunt destinatia unor instructiuni de ramificatie. Cel mai utilizat algoritm euristic în acest sens este cunoscut sub numele de “List Scheduling”. Acesta parcurge graful dependentelor asociat unitatii secventiale de program de jos în sus. În fiecare pas se încearca lansarea în executie a instructiunilor disponibile. Dupa ce aceste instructiuni au fost puse în executie, instructiunile precedente devin disponibile spre a fi lansate în pasul urmator. Fiecarei instructiuni i se ataseaza un grad de prioritate egal cu latenta caii instructiunii. Algoritmul da rezultate cvasioptimale si are avantajul parcurgerii grafului dependentelor de date asociat “basic-block”-ului într-o singura trecere (o prezentare completa se gaseste în [3, 4]). De remarcat ca schedulingul în procesoarele superscalare poate determina o simplificare substantiala a arhitecturii hardware precum si o îmbunatatire a gradului de utilizare aferent resurselor hardware. Optimizarea "basic-block"-urilor unui program nu implica în mod necesar optimizarea întregului program datorita problemelor legate de instructiunile de ramificatie. Reorganizarea programelor care contin branch-uri este mai dificila întrucât aici "mutarile" de instructiuni pot cauza incorectitudini ale programului reorganizat care ar trebui corectate. Aceasta optimizare se mai numeste si optimizare globala. Problema optimizarii globale este una de mare actualitate si interes, întrucât paralelismul la nivelul basic-block-urilor, dupa cum aratau înca din anii '70 pionieri ca *Michael Flynn*, este relativ scazut (doar 2-3 instructiuni). Deoarece majoritatea programelor HLL (High Level Languages) sunt scrise în limbaje imperative si pentru masini secventiale cu un numar limitat de registre în vederea stocarii temporare a variabilelor, este de asteptat ca gradul de dependente între instructiunile adiacente sa fie ridicat. Asadar pentru

marirea nivelului de paralelism este necesara suprapunerea executiei unor instructiuni situate în basic-block-uri diferite, ceea ce conduce la ideea optimizarii globale.

Numeroase studii au aratat ca paralelismul programelor de uz general poate atinge în variante idealizate (resurse hardware nelimitate, redenumire perfecta a registrilor în cazul hazardurilor WAR si WAW, analiza antialias perfecta etc.) în medie 50-60 instructiuni simultane. De remarcat ca schedulerele actuale, cele mai performante, raporteaza performante cuprinse între 3-7 instructiuni simultane. Se apreciaza ca realiste obtinerea în viitorul apropiat a unor performante de 10-15 instructiuni simultane bazat pe îmbunatatirea tehnicilor de optimizare globala. Problema optimizarii globale este una deschisa la ora actuala, având o natura NP - completa. Una dintre cele mai cunoscute metode de optimizare globala este tehnica “Trace Scheduling” (TS). Tehnica TS este similara cu tehnicile de reorganizare în "basic block"-uri, cu deosebirea ca aici se va reorganiza o întreaga cale (Trace) si nu doar un basic - block. În esenta, ea se bazeaza pe optimizarea celor mai probabile cai în a fi executate, bazat pe informatii culese în timpul rularilor (profilings). Se defineste o cale ("Trace") într-un program ce contine salturi conditionate, o ramura particulara a acelui program legata de o asumare data a adreselor acestor salturi. Rezulta deci ca un program care contine n salturi conditionate va avea 2^n posibile cai (trace-uri). Asadar, o cale a unui program va traversa mai multe unitati secventiale din acel program Alte metode de optimizare mai recente, vor fi prezentate sugestiv si pe baza unor implementari academice si comerciale în paragrafele 4 si 5.

O alta tehnica de scheduling esentiala în arhitecturile MEM actuale tine de optimizarea buclelor de program. Aceasta optimizare este foarte importanta pentru ca în buclele de program se pierde cea mai mare parte a timpului de executie aferent respectivului program, dupa regula binecunoscuta care afirma ca "90% din timp executa cca. 10% din program". Aici, 2 tehnici sunt mai uzitate: tehnica “Loop Unrolling” (LU) respectiv tehnica “Software Pipelining” [3]. Tehnica LU se bazeaza pe desfasurarea buclelor de program si apoi, reorganizarea acestora ca basic – block – uri prin algoritmi de tip LS. Prin aceasta desfasurare a buclei de un numar de ori se reduce si din efectul defavorabil al instructiunilor de ramificatie. Tehnica software pipelining (TSP) este utilizata în reorganizarea buclelor de program astfel încât fiecare iteratie a buclei de program reorganizata sa contina instructiuni apartinând unor iteratii diferite din bucla originala. Aceasta reorganizare are drept scop scaderea timpului de executie al buclei prin eliminarea hazardurilor intrinseci, eliminând astfel stagnarile inutile pe timpul procesarii instructiunilor.

Dintre tehnicile agresive de procesare a instructiunilor cele mai în voga actualmente, se numara executia predicativa si respectiv predicativa a instructiunilor, caracteristici esentiale de exemplu în arhitectura IA – 64 Merced (vezi paragraful 5). Executia conditionata (predicativa) se refera la implementarea unor asa numite instructiuni conditionate. O instructiune conditionta se va executa daca o variabila de conditie inclusa în corpul instructiunii îndeplineste conditia dorita. În caz contrar, instructiunea respectiva nu va avea nici un efect (NOP). Variabila de conditie poate fi memorata într-un registru general al procesorului sau în registri special dedicati acestui scop numiti registri booleeni. Astfel de exemplu, instructiunea CMOVZ R1, R2, R3 muta (R2) în R1 daca (R3) = 0. Instructiunea TB5 FB3 ADD R1, R2, R3 executa adunarea numai daca variabilele booleene B5 si B3 sunt '1' respectiv '0'. În caz contrar, instructiunea este inefectiva. Desigur ca variabilele booleene necesita biti suplimentari în corpul instructiunii.

Executia conditionata a instructiunilor este deosebit de utila în eliminarea salturilor conditionate dintr-un program, simplificând programul si transformând deci hazardurile de ramificatie în hazarduri de date. Sa consideram spre exemplificare o constructie *if-then-else* ca mai jos:

```
if (R8<1)                    LT    B6, R8, #1;     if R8<1, B6<---1
  R1 = R2 + R3,              BF    B6, Adr1;       Daca B6=0 salt la Adr1
else                         ADD   R1, R2, R3
  R1 = R5 - R7;              BRA   Adr2 ; salt la Adr2
R10 = R1 + R11;    Adr1:     SUB   R1, R5, R7
                   Adr2:     ADD   R10, R1, R11
```

Prin rescrierea acestei secvente utilizând instructiuni conditionate se elimina cele 2 instructiuni de ramificatie obtinându-se urmatoarea secventa mai simpla si mai eficienta de program:

```
             LT    B6, R8, #1
     TB6     ADD   R1, R2, R3
     FB6     SUB   R1, R5, R7
             ADD   R10, R1, R11
```

Este clar ca timpul de executie pentru aceasta secventa este mai mic decât cel aferent secventei anterioare. Se arata în literatura de specialitate ca astfel de transformari reduc cu cca. 25-30% instructiunile de salt conditionat dintr-un program. Aceasta executie conditionata a instructiunilor faciliteaza executia speculativa a acestora. Codul situat dupa un salt conditionat în program si executat înainte de stabilirea conditiei si adresei de salt cu ajutorul instructiunilor conditionate, se numeste cod cu executie speculativa, operatia respectiva asupra codului numindu-se predicare. Predicarea reprezinta o tehnica de procesare care - utilizând instructiuni cu executie conditionata - urmareste executia paralela prin speculatie a unor instructiuni

si reducerea numarului de ramificatii din program, ambele benefice pt. minimizarea timpului de executie al programului. Acest mod de executie a instructiunilor poate fi deosebit de util în optimizarea executiei unui program.

Prezentam în continuare o secventa de cod initiala si care apoi e transformata de catre scheduler în vederea optimizarii executiei prin speculatia unei instructiuni.

SUB	R1, R2, R3		SUB	R1, R2, R3
LT	B8, R1, #10		LT	B8, R1, #10
BT	B8, Adr	FB8	ADD	R7,R8, R1; speculativa
ADD	R7, R8, R1		BT	B8, Adr
SUB	R10, R7, R4		SUB	R10, R7, R4

Executia speculativa a instructiunii ADD putea fi realizata si în lipsa variabilelor de garda booleene dar atunci putea fi necesara redenumirea registrului R7 (daca acesta ar fi în viata pe ramura pe care saltul se face). Orice instructiune - cu exceptia celor de tip STORE, scrieri în memoria de date - poate fi executata speculativ. O posibila strategie de a permite instructiuni STORE speculative consta în introducerea unui Data Write Buffer (DWB). Memorarea se va face întâi aici si abia când conditia de salt este cunoscuta se va înscrie în memorie. Pe lânga avantajele legate de eliminarea salturilor, facilizarea executiei speculative, predicarii etc., executia conditionata are si câteva dezavantaje dintre care amintim:

- instructiunile conditionate anulate (NOP) necesita totusi un timp de executie. În cazul speculatiei, în aceste conditii performanta în executie scade.
- daca variabila de conditie e evaluata târziu, utilitatea instructiunii conditionate va fi micsorata.
- promovarea unei instructiuni peste mai multe ramificatii conditionate în vederea executiei speculative necesita gardari multiple.
- instructiunile conditionate pot determina scaderea frecventei de tact a microprocesorului.

Având în vedere cele de mai sus, utilitatea executiei conditionate este înca discutata. MIPS, POWER-PC, SUN-SPARC, DEC ALPHA detin doar o instructiune de tip MOVE conditionata, în timp ce alte microarhitecturi precum HEWLET PACKARD PA, HARP, HSA, etc., permit executia conditionata a majoritatii instructiunilor masina. La ora actuala exista înca putine evaluari cantitative care sa stabileasca avantajele/dezavantajele acestei idei într-un mod clar. Implementari concrete ale acestor tehnici, bazat pe exemple, se vor analiza si în paragrafele 4.3.

3. ARHITECTURA SISTEMULUI IERARHIZAT DE MEMORIE

3.1. MEMORII CACHE

Cache: *a safe place for hiding or storing things. (Webster's New World Dictionary of the American Language, Third College Edition - 1988)*

Memoria cache este o memorie situata din punct de vedere logic între CPU (Central Processing Unit - unitate centrala) si memoria principala (uzual DRAM - Dynamic Random Access Memory), mai mica, mai rapida si mai scumpa (per byte) decât aceasta si gestionata – în general prin hardware – astfel încât sa existe o cât mai mare probabilitate statistica de gasire a datei accesate de catre CPU, în cache. Asadar, cache-ul este adresat de catre CPU în paralel cu memoria principala (MP): daca data dorita a fi accesata se gaseste în cache, accesul la MP se aborteaza, daca nu, se acceseaza MP cu penalizarile de timp impuse de latenta mai mare a acesteia, relativ ridicata în comparatie cu frecventa de tact a CPU. Oricum, data accesata din MP se va introduce si în cache.

Memoriile cache sunt implementate în tehnologii de înalta performanta, având deci un timp de acces foarte redus daca sunt integrate în microprocesor (cca. 1 – 5 ns la ora actuala). În prezent presiunea asupra acestor memorii este foarte ridicata, rolul lor fiind acela de a apropia performanta microprocesoarelor (care creste cu cca. 50 – 60 % pe an) cu aceea a memoriilor DRAM, a caror latenta scade cu doar cca. 7 % pe an. În general, pentru a accesa o locatie DRAM, un procesor "pierde" 15 – 50 de impulsuri de tact (~ timp acces DRAM / $T_{CLK,}$ T_{CLK} = perioada ceasului microprocesorului), în schimb accesarea cache-ului se face în doar 1 – 3 impulsuri de tact. Cu alte cuvinte, memoria cache reduce timpul mediu de acces al CPU la MP, ceea ce este foarte util.

Se defineste un acces al CPU cu hit în cache, un acces care gaseste o copie în cache a datei accesate. Un acces cu miss în cache este unul care nu

gaseste o copie în cache a datei accesate de catre CPU si care, prin urmare, adreseaza MP cu toate penalizarile de timp care deriva din accesarea acesteia.

Se defineste ca parametru de performanta al unei memorii cache rata de hit, ca fiind raportul statistic între numarul acceselor cu hit în cache

respectiv numarul total al acceselor CPU la memorie. Masurat pe benchmark-uri (programe de test) reprezentative, la ora actuala sunt frecvente rate de hit de peste 90 %. Rata de miss (RM) este complementara ratei de hit (RH), astfel ca: RH [%] + RM [%] = 100 %. În esenta, utilitatea cache-ului deriva din urmatorul fapt: la o citire cu miss (din MP), data adusa din MP este introdusa si în cache, în speranta ca la o urmatoare citire a aceleiasi date, aceasta se va gasi în cache (hit). În realitate, în cazul unei citiri cu miss în cache se aduce din MP nu doar data (cuvântul) dorita de catre CPU ci un întreg bloc (4 – 16 cuvinte) care evident contine data respectiva. O citire cu miss presupune aducerea blocului din MP dar înainte de aceasta se impune evacuarea în MP a unui bloc din cache. Asadar, transferul din cache în MP se face tot la nivel de bloc si nu de cuvânt. Astfel, se optimizeaza traficul între cache si MP pe baza a 2 principii care vor fi discutate în continuare.

În esenta, eficienta memoriilor cache se bazeaza pe 2 principii de natura statistica si care caracterizeaza intrinsec notiunea de program: principiile de localitate temporala si spatiala. Conform principiului de localitate temporala, exista o mare probabilitate ca o data (instructiune) accesata acum de catre CPU sa fie accesata din nou, în viitorul imediat. Conform principiului de localitate spatiala, exista o mare probabilitate ca o data situata în imediata vecinatate a unei date accesate curent de catre CPU, sa fie si ea accesata în viitorul apropiat (pe baza acestui principiu statistic se aduce din MP în cache un întreg bloc si nu doar strict cuvântul dorit de catre CPU). O bucla de program – structura esentiala în orice program – exemplifica foarte clar aceste principii si justifica eficienta conceptului de cache.

O combinare a celor 2 principii anterior expuse conduce la celebra "regula 90/10" care spune ca cca. 90 % din timpul de rulare al unui program se executa doar cca. 10 % din codul acestuia. Personal, credem ca mai putin. Pe baza acestor principii empirice se situeaza întreg esafodajul conceptului de cache; eficienta sa deosebita nu poate fi explicata prin considerente analitice pentru simplul fapt ca este practic imposibil a descrie analitic notiunea de program. În fond, ce este un program? Care este distributia instructiunilor sau a primitivelor structurale într-un program? Poate fi aceasta descrisa concret, pe baza unor modele deterministe sau aleatoare? Dificultatea unor raspunsuri exacte la aceste întrebari – data în fond de

imposibilitatea punerii în ecuatie a mintii umane, cea care creeaza infinita diversitate de “programe” – face ca cea mai buna explicatie asupra eficientei memoriilor cache sa stea în cele 2 principii empirice anterior schitate, caracterizând intrinsec notiunea de program.

Din punct de vedere arhitectural, exista 3 tipuri distincte de memorii cache în conformitate cu gradul de asociativitate: cu mapare directa, semiasociative si total asociative.

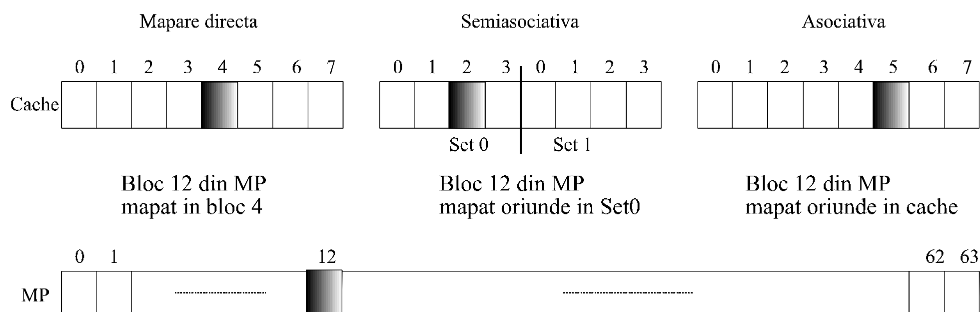


Figura 3.1. Scheme de mapare în cache

La cache-urile cu mapare directa, ideea principala consta în faptul ca un bloc din MP poate fi gasit în cache (hit) într-un bloc unic determinat. În acest caz regula de mapare a unui bloc din MP în cache este:

(**Adresa bloc MP**) modulo (**Nr. blocuri din cache**)

Strictetea regulii de mapare conduce la o simplitate constructiva a acestor memorii dar si la fenomenul de interferenta al blocurilor din MP în cache. Astfel, de exemplu, blocurile 12, 20, 28, 36, 42 etc. nu pot coexista în cache la un moment dat întrucât toate se mapeaza pe blocul 4 din cache. Prin urmare, o bucla de program care ar accesa alternativ blocurile 20 si 28 din MP ar genera o rata de hit egala cu zero.

La cache-urile semiasociative exista mai multe seturi, fiecare set având mai multe blocuri componente. Aici, regula de mapare precizeaza strict doar setul în care se poate afla blocul dorit, astfel:

(**Adresa bloc MP**) modulo (**Nr. seturi din cache**)

În principiu blocul dorit se poate mapa oriunde în setul respectiv. Mai precis, la un miss în cache, înainte de încarcarea noului bloc din MP, trebuie evacuat un anumit bloc din setul respectiv. În principiu exista implementate doua-trei tipuri de algoritmi de evacuare: pseudorandom (cvasialeator),

FIFO si LRU ("Least Recently Used"). Algoritmul LRU evacueaza blocul din cache cel mai demult neaccesat, în baza principiului de localitate temporala (aflat oarecum în contradictie cu o probabilistica markoviana care ar sugera sa fie pastrat!). Daca un set din cache-ul semiasociativ contine N blocuri atunci cache-ul se mai numeste "tip N-way set associative".

Este evident ca într-un astfel de cache rata de interferenta se reduce odata cu cresterea gradului de asociativitate (N "mare"). Aici, de exemplu, blocurile 12, 20, 28 si 36 pot coexista în setul 0. Prin reducerea posibilelor interferente ale blocurilor, cresterea gradului de asociativitate determina îmbunatatirea ratei de hit si deci a performantei globale. Pe de alta parte însa, asociativitatea impune cautarea dupa continut (se cauta deci într-un set daca exista memorat blocul respectiv) ceea ce conduce la complicatii structurale si deci la cresterea timpului de acces la cache si implicit la diminuarea performantei globale. Optimizarea gradului de asociativitate, a capacitatii cache, a lungimii blocului din cache etc., nu se poate face decât prin laborioase simulari software, variind toti acesti parametrii în vederea minimizarii ratei globale de procesare a instructiunilor [instr./cicli].

În fine, memoriile cache total asociative, implementeaza practic un singur set permitând maparea blocului practic oriunde în cache. Ele nu se implementeaza deocamdata în siliciu datorita complexitatii deosebite si a timpului prohibit de cautare. Reduc însa (practic) total interferentele blocurilor la aceeasi locatie cache si constituie o metrica superioara utila în evaluarea ratei de hit pentru celelalte tipuri de cache-uri (prin comparatie). Cele 3 scheme urmatoare prezinta implementari realizate pentru tipurile de cache anterior discutate.

Cache semiasociativ pe 2 cai

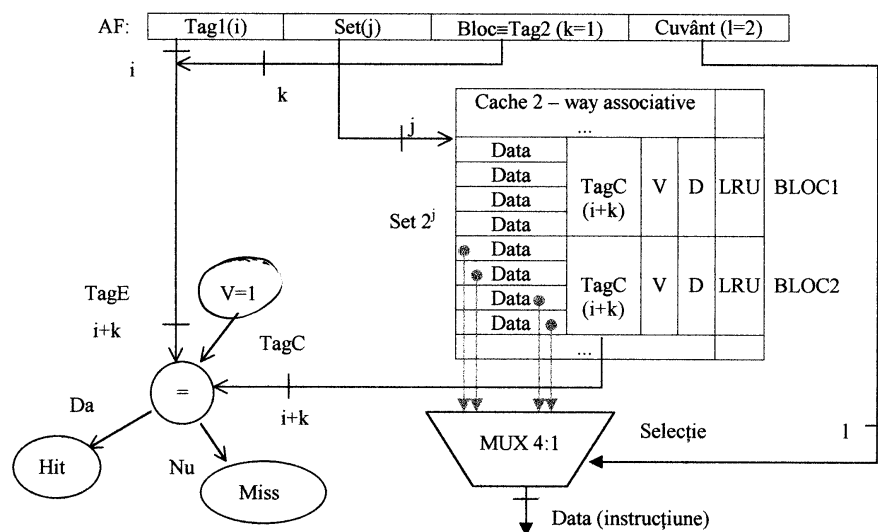


Figura 3.2. Cache semiasociativ pe 2 cai

Cache complet associativ

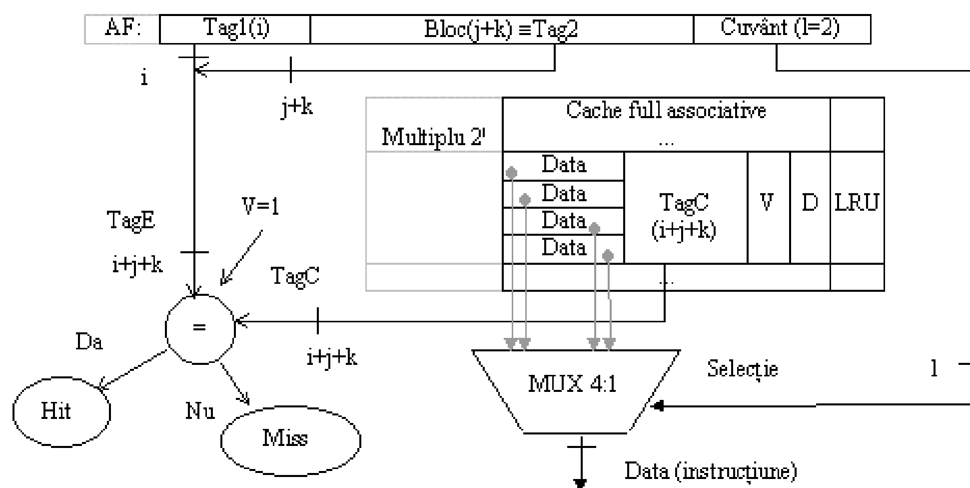


Figura 3.3. Cache complet associativ

Cache direct mapat

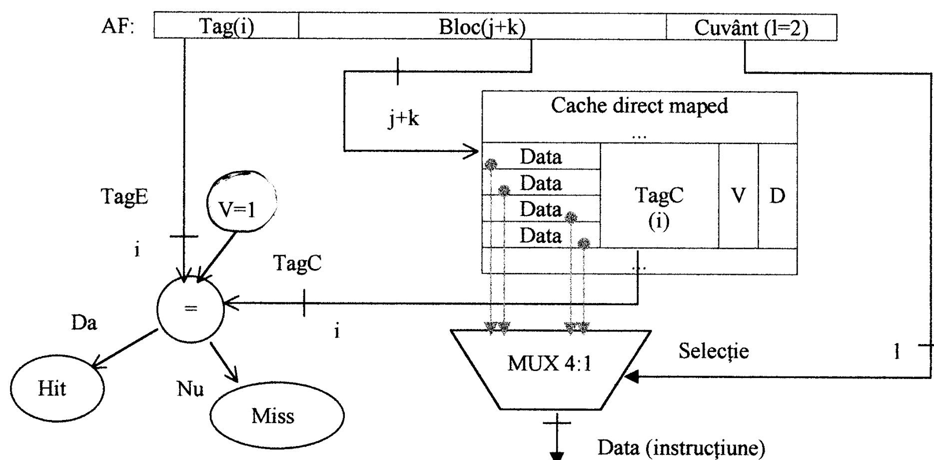


Figura 3.4. Cache direct mapat

S-a considerat un bloc compus din 4 cuvinte. Bitul V este un bit de validare a blocului, V = 1 fiind o conditie necesara a obtinerii hitului. Bitul este util îndeosebi în Sistemele multiprocesor în vederea mentinerii coerentei memoriilor cache locale datorita redundantei informationale. Mai precis, aici apare necesitatea citirii din cache-ul propriu a ultimei copii modificate a datei respective. Când un procesor modifica o copie locala a unei date, toate blocurile care contin acea data din cadrul celorlalte procesoare, trebuie invalidate prin resetarea V = 0. Necesitatea invalidarii blocurilor (V = 0) apare chiar si în sistemele uniprocesor. Imediat dupa resetarea sistemului, uzual, procesorul executa un program încarcator rezident în memoria EPROM. Cum imediat dupa initializarea sistemului continutul cache-ului e practic aleator, pentru a evita false hituri la citirea programului încarcator din EPROM, se initializeaza bitii V cu zero. La prima încarcare a unei date (instructiuni) în cache, bitul V aferent se va seta pe ‘1’, validând astfel hitul.

Bitul D (Dirty) este pus pe ‘0’ la încarcarea initiala a blocului în cache. La prima scriere a acelui bloc, bitul se pune deci pe ‘1’. Evacuarea propriu-zisa a blocului se face doar daca bitul D = 1. Practic prin acest bit se minimizeaza evacuarile de blocuri în MP, pe baza principiului ca un bloc trebuie evacuat numai daca a fost scris în cache.

În acest sens, din punct de vedere al acceselor de scriere a unui procesor, exista 2 posibilitati:

a) Strategia "Write Through" (WT), prin care informatia este scrisa de catre procesor atât în blocul aferent din cache cât si în blocul corespunzator din memoria principala.
b) Strategia "Write - Back" (WB), prin care informatia este scrisa numai în cache, blocul modificat fiind transferat în MP numai la evacuarea din cache.

În vederea mentinerii coerentei cache-urilor cu precadere în sistemele multimicroprocesor – exista 2 posibilitati în functie de ce se întâmpla la o scriere (vezi pentru detalii capitolul dedicat sistemelor multimicroprocesor):

a) Write invalidate – prin care CPU care scrie determina ca toate copiile din celelalte memorii cache sa fie invalidate înainte ca el sa-si modifice blocul din cache-ul propriu.
b) Write Broadcast – CPU care scrie pune data de scris pe busul comun spre a fi actualizate toate copiile din celelalte cache-uri.

Ambele strategii de mentinere a coerentei pot fi asociate cu oricare dintre protocoalele de scriere (WT, WB) dar de cele mai multe ori se prefera WB cu invalidare. Nu detaliem aici problemele de coerenta întrucât acestea se refera cu deosebire la problematica sistemelor multiprocesor si deci depasesc cadrul acestei prezentari.

Apar posibile 4 procese distincte într-un cache ca în tabelul urmator:

Tip acces	**Hit / Miss**	**Actiune în cache**
Citire	Miss	Evacuare bloc + încarcare bloc nou
Citire	Hit	(comparare tag-uri)
Scriere	Miss	Evacuare bloc + încarcare bloc nou + scriere data în bloc
Scriere	Hit	Scriere data în blocul din cache (WB)

Tabelul 3.1.

Tipuri de acces în cache

Asadar, memoriile cache îmbunatatesc performanta îndeosebi pe citirile cu hit iar în cazul utilizarii scrierii tip "Write Back" si pe scrierile cu hit.

Îmbunatatirea accesului la memorie pe citirile CPU este normala având în vedere ca acestea sunt mult mai frecvente decât scrierile (orice instructiune implica cel putin o citire din memorie pentru aducerea sa; statistic, cca. 75 % din accesele la memorie sunt citiri).

Explicatia la cauzele miss-urilor în cache-uri, conform literaturii acestui domeniu, sunt de 3 tipuri:

- datorita faptului ca în fond primul acces la un bloc genereaza întotdeauna miss (*compulsory*); sunt inevitabile.
- datorita capacitatii fatalmente limitate a cache-ului care nu poate contine la un moment dat toate blocurile din MP, ceea ce implica evacuari / încarcari (*capacity*).
- datorita interferentelor (conflictelor) unor blocuri din MP pe acelasi bloc din cache (*conflict*); acestea se reduc odata cu cresterea capacitatii si a gradului de asociativitate.

Primul care a pus în lumina conceptul de memorie cache a fost prof. Maurice Wilkes (Univ. Cambridge, Anglia) – un pionier al calculatoarelor care a inventat în 1951 si tehnica microprogramarii unitatilor de comanda aferente procesoarelor – într-un articol publicat în 1965 ("*Slave memories and dynamic storage allocation*", IEEE Trans. Electronic Computers, April, 1965). Prima implementare a unui cache (cu mapare directa) apartine probabil lui Scarrott, în cadrul unui sistem experimental construit tot la Universitatea din Cambridge. Primul sistem comercial care utiliza cache-urile a fost IBM 360/85 (1968). Conceptul de cache s-a dovedit a fi foarte fecund nu numai în hardware dar si în software prin aplicatii dintre cele mai diverse în sistemele de operare (memoria virtuala), retele de calculatoare, baze de date, compilatoare etc.

Pentru a reduce rata de miss a cache-urilor mapate direct (fara sa se afecteze însa timpul de hit sau penalitatea în caz de miss), cercetatorul Norman Jouppi (DEC) a propus conceptul de "victim cache". Aceasta reprezinta o memorie mica complet asociativa, plasata între primul nivel de cache mapat direct si memoria principala. Blocurile înlocuite din cache-ul principal datorita unui miss sunt temporar memorate în victim cache. Daca sunt referite din nou înainte de a fi înlocuite din victim cache, ele pot fi extrase direct din victim cache cu o penalitate mai mica decât cea a memoriei principale [5]. Deoarece victim cache-ul este complet asociativ, multe blocuri care ar genera conflict în cache-ul principal mapat direct, ar putea rezida în victim cache fara sa dea nastere la conflicte. Decizia de a plasa un bloc în cache-ul principal sau în victim cache (în caz de miss) este facuta cu ajutorul unei informatii de stare asociate blocurilor din cache. Bitii de stare contin informatii despre istoria blocului. Aceasta idee a fost propusa de McFarling, care foloseste informatia de stare pentru a exclude blocurile sigure din cache-ul mapat direct, reducând înlocuirile ciclice implicate de acelasi bloc. Aceasta schema, numita excludere dinamica, reduce miss-urile de conflict în multe cazuri. O predictie gresita implica un acces în nivelul urmator al ierarhiei de memorie contrabalansând eventuale câstiguri în

performanta. Schema este mai putin eficace cu blocuri mari, de capacitati tipice cache-urilor microprocesoarelor curente.

Pentru a reduce numarul de interschimbari dintre cache-ul principal si victim cache, Stiliadis si Varma au introdus un nou concept numit selective victim cache(SVC)[6].

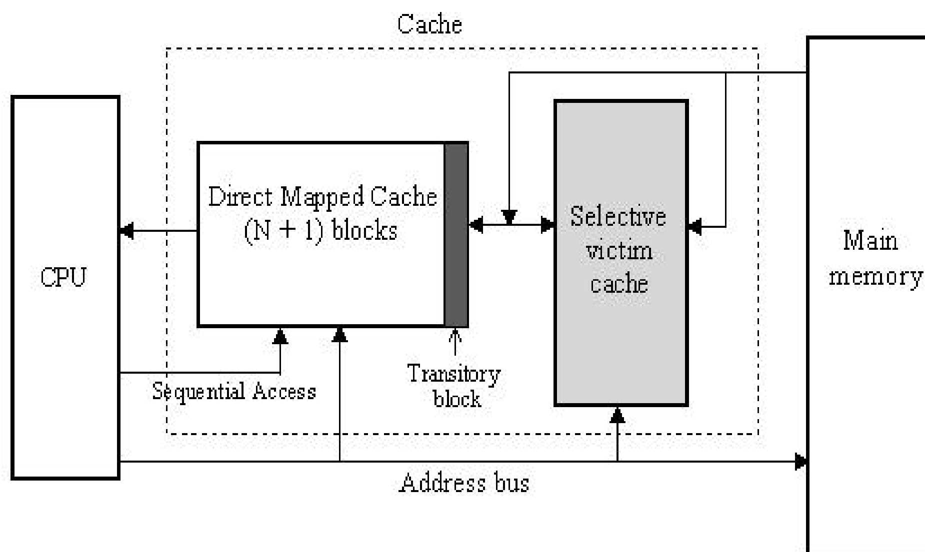


Figura 3.5. Ierarhia de memorie pentru scema cu Selective Victim Cache

Cu SVC, blocurile aduse din memoria principala sunt plasate selectiv fie în cache-ul principal cu mapare directa fie în selective victim cache, folosind un algoritm de predictie euristic bazat pe istoria folosirii sale. Blocurile care sunt mai putin probabil sa fie accesate în viitor sunt plasate în SVC si nu în cache-ul principal. Predictia este de asemenea folosita în cazul unui miss în cache-ul principal pentru a determina daca este necesara o schimbare a blocurilor conflictuale. Algoritmul obiectiv este de a plasa blocurile, care sunt mai probabil a fi referite din nou, în cache-ul principal si altele în victim cache.

La referirea unui cache mapat direct, victim cache-ul este adresat în paralel; daca rezulta miss în cache-ul principal, dar hit în victim cache, instructiunea (în cazul ICache-ului) este extrasa din victim cache. Penalitatea pentru miss în cache-ul principal, în acest caz este mult mai redusa decât costul unui acces în nivelul urmator de memorie. Algoritmul de victim cache încearca sa izoleze blocurile conflictuale si sa le memoreze doar unul în cache-ul principal restul în victim cache. Daca numarul blocurilor conflictuale este suficient de mic sa se potriveasca în victim cache, atât rata de miss în nivelul urmator de memorie cât si timpul mediu

de acces va fi îmbunatatit datorita penalitatii reduse implicate de prezenta blocurilor în victim cache.

Cache-ul mapat direct creste cu un bloc pentru a implementa conceptul de selective victim cache. Acest bloc aditional se numeste *bloc tranzitoriu*, si este necesar pentru doua motive. Primul ar fi acela ca, blocul tranzitoriu este folosit de algoritmul de predictie pentru referiri secventiale într-un acelasi bloc. Hardware-ul este capabil sa determine accese secventiale, folosind semnalul "Acces Secvential" activat de CPU, când referirea curenta se face în acelasi bloc ca si cel anterior. Semnalul este folosit de catre cache pentru a evita actualizarea bitilor de stare folositi de algoritmul de predictie la referinte repetate în acelasi bloc tranzitoriu. Al doilea motiv consta în faptul ca, atunci când are loc un hit în victim cache si algoritmul de predictie decide sa nu se interschimbe blocurile, blocul corespondent este copiat din victim cache în blocul tranzitoriu. Astfel, blocul tranzitoriu serveste ca buffer, accesele secventiale la acel bloc fiind satisfacute direct din acest buffer la timpul de acces al cache-ului principal. Similar, la un miss în urmatorul nivel de memorie, algoritmul de predictie va decide sa plaseze blocul sosit în victim cache si în blocul tranzitoriu.

Întrucât un al doilea sau un al n-lea acces consecutiv în acelasi bloc în cache-ul principal poate fi servit din blocul tranzitoriu, acestuia îi este adaugat un bit de stare pentru a adresa cache-ul principal. Acest bit de stare urmareste starea datei din blocul tranzitoriu. Când starea este *normala*, adresa sosita pe bus este decodificata pentru a accesa cache-ul principal în mod obisnuit; când starea este *speciala*, accesul se face în blocul tranzitoriu. Figura urmatoare arata tranzitiile dintre cele doua stari. Mai jos se prezinta acest algoritm sub forma de "masina secventiala de stare".

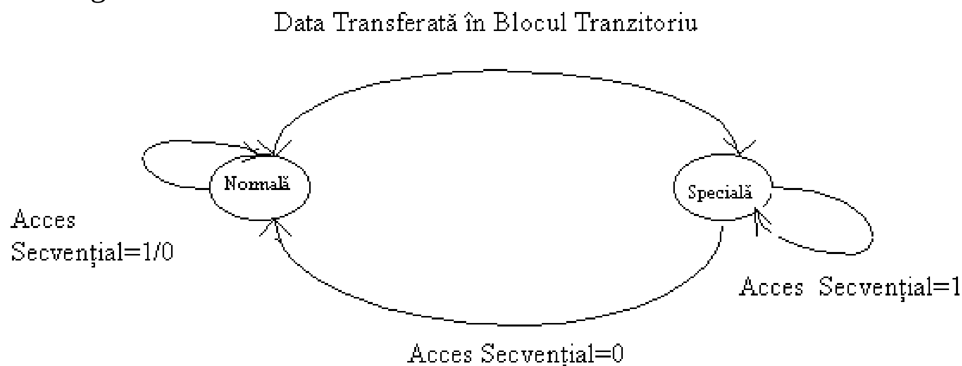


Figura 3.6. Masina secventiala de stare si tranzitiile ei

Initial starea masinii este resetata în stare *normala*. Daca avem un miss în cache-ul mapat direct, acesta este servit fie de victim cache fie de

nivelul urmator de memorie. În fiecare din cazuri, algoritmul de predictie este folosit pentru a determina care bloc urmeaza a fi memorat în cache-ul principal. Daca algoritmul de predictie plaseaza blocul accesat în cache-ul principal, starea masinii ramâne în stare normala. Altfel, blocul este copiat în blocul tranzitoriu di acest cache si masina tranziteaza în starea *speciala*. Referirea secventiala a aceluiasi bloc pastreaza semnalul "Acces Secvential" activat iar masina în starea *speciala*. Datele se extrag din blocul tranzitoriu. Primul acces nesecvential reseteaza starea masinii în stare *normala*, distingându-se trei cazuri distincte, pe care le vom discuta mai jos.

Algoritmul Selective Victim Cache

1. ***Hit în cache-ul principal***: daca cuvântul este gasit în cache-ul principal, el este extras pentru CPU. Nu este nici o diferenta fata de cazul cache-ului mapat direct. Singura operatie suplimentara este o posibila actualizare a bitilor de stare folositi de schema de predictie. Actualizarea se poate face în paralel cu operatia de fetch si nu introduce întârzieri suplimentare.
2. ***Miss în cache-ul principal, hit în victim cache***: în acest caz, cuvântul este extras din victim cache în cache-ul mapat direct si înaintat CPU. Un algoritm de predictie este invocat pentru a determina daca va avea loc o interschimbare între blocul referit si blocul conflictual din cache-ul principal. Daca algoritmul decide ca blocul din victim cache este mai probabil sa fie referit din nou decât blocul conflictual din cache-ul principal se realizeaza interschimbarea; altfel blocul din victim cache este copiat în blocul tranzitoriu al cache-ului principal iar masina secventiala de stare trece în starea speciala. Data poate fi înaintata CPU. În ambele cazuri blocul din victim cache este marcat drept cel mai recent folosit din lista LRU. În plus, bitii de predictie sunt actualizati pentru a reflecta istoria acceselor.
3. ***Miss atât în cache-ul principal cât si în victim cache***: daca cuvântul nu este gasit nici în cache-ul principal nici în victim cache, el trebuie extras din nivelul urmator al ierarhiei de memorie. Aceasta înseamna ca fie blocul corespondent din cache-ul principal este "gol", fie noul bloc este în conflict cu un alt bloc memorat în cache (mai probabil). În primul caz, noul bloc este adus în cache-ul principal iar victim cache-ul nu este afectat. În cel de-al doilea caz, trebuie aplicat algoritmul de predictie pentru a determina care din blocuri este mai probabil sa fie referit pe viitor. Daca blocul care soseste din memoria centrala are o probabilitate mai mare decât blocul conflictual din cache-ul principal, ultimul este mutat în victim cache si noul bloc îi ia locul în cache; altfel, blocul sosit este directionat spre victim cache si copiat în blocul tranzitoriu al cache-

ului mapat direct, de unde poate fi accesat mai iute de catre CPU. Masina secventiala de stare trece în starea speciala iar bitii de predictie sunt actualizati.

Diferenta de esenta dintre schema prezentata (*selective victim cache*) si conceptul de victim cache simplu se observa în cazurile 2 si 3. În cazul 2, blocurile conflictuale din cache-ul principal si cele din victim cache sunt întotdeauna schimbate în cazul folosirii victim cache-ului traditional, pe când schema prezentata face acest lucru într-un mod selectiv, euristic. Similar, în cazul 3, prin folosirea victim cache-ului obisnuit blocurile din memorie sunt întotdeauna plasate în cache-ul principal, pe cînd în cazul *Selective Victim Cache*-ului plaseaza aceste blocuri selectiv în cache-ul principal sau în victim cache. Orice algoritm de înlocuire poate fi folosit pentru victim cache. LRU (cel mai putin recent referit) pare sa fie cea mai buna alegere, întrucât scopul victim cache-ului este de a captura cele mai multe victime recent înlocuite si victim cache-ul este de dimensiune mica.

Algoritmul de Predictie

Scopul algoritmului de predictie este de determina care din cele doua blocuri conflictuale este mai probabil sa fie referit pe viitor. Blocul considerat cu o probabilitate mai mare de acces în viitor este plasat în cache-ul principal, celalalt fiind plasat în victim cache. Astfel, daca blocul din victim cache este pe viitor înlocuit datorita capacitatii reduse a victim cache-ului, impactul ar fi mai putin sever decât alegerea opusa (interschimbarea permanenta a blocurilor din cazul schemei cu victim cache obisnuit).

Algoritmul de predictie se bazeaza pe algoritmul de excludere dinamica propus de McFarling [7]. Algoritmul foloseste doi biti de stare asociati fiecarui bloc, numiti *hit bit* si *sticky bit*. Hit bit este asociat logic cu blocul din nivelul 1 (L1 - level one) al cache-ului care se afla pe nivelul 2 (L2) sau în memoria centrala. Hit bit egal cu 1 logic indica, faptul ca a avut cel putin un acces cu hit la blocul respectiv de cînd el a parasit cache-ul principal (cache-ul de pe nivelul L1). Hit bit egal cu 0 înseamna ca blocul corespunzator nu a fost deloc accesat de când a fost înlocuit din cache-ul principal. Într-o implementare ideala, bitii de hit sunt mentinuti în nivelul L2 de cache sau în memoria principala si adusi în nivelul L1 de cache cu blocul corespondent. Daca blocul este înlocuit din cache-ul principal (L1 cache), starea bitului de hit trebuie actualizata în L2 cache sau în memoria centrala. Când un bloc, sa-l numim α, a fost adus în cache-ul principal, bitul sau sticky este setat. Fiecare secventa cu hit la blocul α reîmprospateaza bitul sticky la valoarea 1. La referirea unui bloc conflictual, fie acesta β, daca algoritmul de predictie decide ca blocul sa nu fie înlocuit din cache-ul

principal atunci bitul sticky este resetat. Daca un acces ulterior în cache-ul principal intra în conflict cu blocul care are bitul sticky resetat, atunci blocul va fi înlocuit din cache-ul principal. De aceea, sticky bit de valoare 1 pentru blocul α semnifica faptul ca nu a avut loc nici o referire la un bloc conflictual cu α, de la ultima referire a acestuia.

Este usor de înteles rolul blocului tranzitoriu în algoritmul de predictie. Daca algoritmul trateaza toate fetch-urile în acelasi fel, accesele secventiale în acelasi bloc vor seta întotdeauna bitul sticky. Algoritmul de predictie va fi incapabil sa determine daca blocul a fost referit repetat în interiorul unei bucle, sau daca mai mult decât un cuvânt din acelasi bloc a fost extras din cache fara o referinta intervenita la un alt bloc. O solutie similara a acestei probleme a fost propusa si în [7].

```
Cazul 1: Accesarea blocului β, hit în cache-ul principal
        Actualizează biții hit și sticky
                hit[β]←1;sticky[β]←1;
Cazul 2: Accesarea blocului β, hit în victim cache
        Fie α blocul conflictual din cache-ul principal
                if sticky[α] = 0 then
                        interschimbă α cu β
                        sticky[β]←1; hit[β]←1
                else
                        if hit[β] = 0 then
                                sticky[α]←0;
                                copiază β în blocul tranzitoriu
                        else
                                interschimbă α cu β
                                sticky[β]←1; hit[β]←0;
                        endif
                endif
Cazul 3: Accesarea blocului β, miss atât în cache-ul principal cât și în
victim cache
        α este blocul conflictual din cache-ul principal
                if sticky[α] = 0 then
                        mută α în victim cache
                        transferă β în cache-ul principal
                        sticky[β]←1;  hit[β]←1;
                else
                        if hit[β] = 0 then
                                transferă β în victim cache
                                copiază β în blocul tranzitoriu
                                sticky[α]←0;
                        else
                                mută α în victim cache
                                transferă β în cache-ul principal
                                sticky[β]←1; hit[β]←0;
                        endif
                endif
```

Figura 3.7. Algoritmul de Selective Victim Cache

În algoritmul Selective Victim Cache prezentat în figura anterioara, se disting trei cazuri: în primul caz, un hit în cache-ul principal seteaza bitii de stare hit si sticky. În al doilea caz, blocul accesat, fie acesta β, se considera rezident în victim cache. Acesta implica un conflict între blocul β si cel din cache-ul principal, notat α. În acest caz, algoritmul de predictie este aplicat pentru a determina daca va avea loc o interschimbare. Daca bitul sticky al lui α este 0, semnificând faptul ca blocul nu a fost accesat de la conflictul anterior la acest bloc, noul bloc β primeste o prioritate superioara lui α, determinând o interschimbare. De asemenea, daca bitul hit al lui β este setat pe 1, acestuia îi este data o prioritate mai mare decât lui α, si ele sunt interschimbate. Daca bitul sticky al lui α este 1 si bitul hit al lui β este 0, accesul este satisfacut din victim cache si nu are loc nici o interschimbare (se considera ca blocul β nu este suficient de "valoros" pt. a fi adus în cache-ul principal). Bitul sticky aferent lui α este resetat astfel încât o secventa urmatoare care implica conflict la acest bloc va determina mutarea lui α din cache-ul principal. În final, cazul 3 al algoritmului prezinta secventa de actiuni care au loc în cazul unor accese cu miss atât în cache-ul principal cât si în victim cache. Secventa este similara cu cea de la cazul 2, cu exceptia faptului ca, destinatia blocului sosit se alege fie cache-ul principal fie victim cache-ul. În situatia cu victim cache simplu, blocul conflictual din cache-ul principal era mutat în victim cache înainte sa fie înlocuit. În cazul de fata când blocul sosit este plasat în victim cache, el este de asemenea plasat si în blocul tranzitoriu pentru a servi eventualele viitoare referinte secventiale.

Operatiile algoritmului de Selective Victim Cache pot fi ilustrate printr-o secventa de instructiuni repetate $(\alpha^m\beta\gamma)^n$ implicând trei blocuri conflictuale α, β si γ. Notatia $(\alpha^m\beta\gamma)^n$ reprezinta executia unei secvente compusa din doua bucle de program imbricate, bucla interioara constând în *m* referinte la blocul α, urmate de accesul la blocurile β si γ în bucla exterioara, care se executa de *n* ori. Primul acces îl aduce pe α în cache-ul principal si atât bitul hit cât si cel sticky sunt setati dupa cel mult doua referiri ale acestuia. Când β este referit, bitul sau hit este initial 0. De aceea el nu-l înlocuieste pe α în cache-ul principal si este memorat în victim cache. Conflictul generat determina resetarea bitului sticky al lui α. Când γ este referit, bitul sau hit este 0, dar bitul sticky al lui α este tot 0. Deci, γ îl înlocuieste pe α. Blocul α este transferat în victim cache si bitul sau hit ramâne 1 datorita referintei sale anterioare. În ciclul urmator când α este referit din nou, el este mutat înapoi în cache-ul principal datorita bitului sau de hit, ramas setat. Astfel, daca victim cache-ul este suficient de mare pentru

a încape atât α si β, sau β si γ, doar trei referinte ar fi servite de catre al doilea nivel de cache. Numarul total de interschimbari nu va depasi $2n$. În cazul unei scheme simple de predictie fara victim cache, numarul total de referiri cu miss ar fi $2n$, în cazul în care schema poate rezolva doar conflicte între doua blocuri. Un victim cache simplu, fara predictie ar fi capabil sa reduca numarul de accese cu miss la cel de-al doilea nivel de cache la 3, dar ar necesita $3n$ interschimbari în timpul executiei buclei exterioare, cu influente evident defavorabile asupra timpului global de procesare. Aceasta arata avantajul Selective Victim Cache-ului superioara altor scheme care trateaza conflicte implicând mai mult de doua blocuri. De retinut ca, penalitatea pentru o predictie gresita în aceasta schema este limitata la accesul în victim cache si o posibila interschimbare, presupunând ca victim cache-ul este suficient de mare pentru a retine blocurile conflictuale între accese.

Metrici de performanta

Metricile de performanta folosite sunt rata de miss la nivelul L1 de cache si timpul mediu de acces la ierarhia de memorie. În cazurile cu victim cache simplu si cel cu victim cache selectiv, folosim de asemenea si numarul de interschimbari între cache-ul principal si victim cache ca metrica de comparatie. Rata de miss la nivelul L1 de cache se bazeaza pe numarul de referinte propagate în nivelul urmator al ierarhiei de memorie. În caz de miss în cache-ul principal acestea sunt servite de catre victim cache, fiind platita o penalitate pentru accesul în victim cache precum si pentru interschimbarile de blocuri rezultate între cache-ul principal si victim cache. Timpul mediu de acces la ierarhia de memorie ia în calcul si aceste penalizari si de aceea este un bun indicator al performantei memoriei sistemului, desigur mai complet decât rata de miss în nivelul L1 de cache.

Deoarece obiectivul principal al victim cache-ului este de a reduce numarul de *miss-uri de conflict* în cache-ul mapat direct, este de asemenea important sa comparam procentul de miss-uri de conflict eliminate prin fiecare din scheme. Miss-urile de conflict sunt de obicei calculate ca miss-uri suplimentare ale unui cache, comparate cu un cache complet asociativ de aceeasi marime si care dezvolta un acelasi algoritm de înlocuire. Algoritmul folosit este LRU (Least Recently Used, cel mai de demult nefolosit) [8, 9]. Desi acest model pare intuitiv corect, el poate genera si rezultate eronate uneori. De exemplu, numarul total de accese cu miss poate uneori sa creasca când creste asociativitatea, iar politica de înlocuire LRU este departe de a fi cea optima pentru unele din programe. Aceasta anomalie poate fi evitata prin folosirea algoritmului optim (OPT) în loc de LRU ca baza pentru clasificarea miss-urilor în cache [10]. Algoritmul OPT, prima data studiat în

[11], înlocuieste întotdeauna blocul care va fi înlocuit cel mai târziu în viitor. Un astfel de algoritm s-a dovedit a fi optimal pentru toate pattern-urile de program, ratele de miss fiind cele mai mici în acest caz, dintre toate politicile de înlocuire folosite.

Modelarea timpului de acces la ierarhia de memorie

Estimarea timpului de acces se face ca o functie de marimea cache-ului, dimensiunea blocului, asociativitatea si organizarea fizica a cache-ului. Presupunem ca penalitatea medie pentru un miss în cache-ul de pe nivelul L1 este acelasi pentru toate arhitecturile si este de *p* ori ciclul de baza al cache-lui principal, unde *p* variaza între 1 si 100. Consideram un bloc de dimensiune de 32 octeti, penalitate în caz de miss de 10-50 de cicli, în caz ca nu exista un nivel L2 de cache. Câteva studii, precum [12] raporteaza ca penalitatea pentru un miss poate fi pâna la 100-200 de cicli când nu este inclus un al doilea nivel de cache.

Parametrii	**Cache Mapat Direct**	**Victim Cache Simplu**	**Selective Victim Cache**	**2-way cache**
Referinte totale	R			
Numar total de miss-uri în L1 cache	M_d	M_v	M_s	M_2
Hit-uri în victim cache		h_v	h_s	
Interschimbari între victim cache si cache-ul principal		I_v	I_s	
Timp mediu de acces	T_d	T_v	T_s	T_2
Timp mediu de penalizare (în cicli CPU)	p			
Perioada de tact CPU	clk			$clk_{2\text{-way}}$

Tabelul 3.2.

Notatiile folosite în calculul timpului de acces

Tabelul 3.2., rezuma toate notatiile privitoare la calculul timpului de acces la memorie. *R* este numarul total de referinte generate de programele de tip trace. În cazul cache-ului simplu mapat direct, M_d reprezinta numarul total de accese cu miss în cache. În cazul folosirii unui victim cache obisnuit sau a unui Selective Victim Cache, M_v si M_s sunt folosite pentru a nota numarul de accese cu miss în primul nivel de cache care sunt servite de al doilea nivel al ierarhiei de memorie. Numarul total de hituri în victim cache pentru aceste scheme le-am notat cu h_v si respectiv h_s.

Timpul mediu de acces pentru un cache mapat direct se calculeaza astfel:

$$T_d = clk \times \frac{R + p \times M_d}{R} = clk\left(1 + p \cdot \frac{M_d}{R}\right) \qquad (3.1)$$

Pentru fiecare miss, sunt necesari *p* cicli suplimentari. Presupunem ca cei *p* cicli includ toate "cheltuielile" CPU-ului. În cazul victim cache-ului simplu, o penalitate de *p* cicli este produsa la fiecare miss în primul nivel al ierarhiei de memorie. În plus, pentru fiecare referinta servita de catre victim cache, o penalizare suplimentara de cel putin un ciclu este platita pentru accesarea în victim cache, iar operatia de interschimbare dintre cache-ul principal si victim cache necesita o penalitate de câtiva cicli (presupunem 3 cicli, de altfel minimali). Aceasta penalitate ar fi chiar mai mare daca matricea memoriei cache-ului mapat direct sau a victim cache-ului este organizata fizic în cuvinte egale cu o fractiune din marimea blocului de cache. De exemplu, blocul poate avea dimensiunea de 32 de octeti, dar cache-ul poate fi organizat în cuvinte de 8 sau 16 octeti. În acest caz penalitatea pentru interschimbare va fi multiplicata cu raportul:

$$\frac{\text{Dimensiunea blocului de cache}}{\text{Mãrimea cuvântului de date al cache-ului}}$$

Astfel, timpul mediu de acces la memorie pentru un sistem cu victim cache simplu, se calculeaza astfel:

$$T_v = clk\left(1 + p \cdot \frac{M_v}{R} + \frac{h_v + 3 \times I_v}{R}\right) \qquad (3.2)$$

Într-un sistem cu *Selective Victim Cache*, timpul mediu de acces la memorie poate fi calculat în acelasi fel ca în cazul victim cache-ului simplu. O penalitate de *p* cicli este aplicata de câte ori este accesat nivelul urmator al ierarhiei de memorie. Un ciclu suplimentar este necesar la un hit în victim cache si 3 cicli suplimentari pentru operatia de interschimbare de blocuri. Timpul mediu de acces la memorie este dat de formula:

$$T_s = clk\left(1 + p \cdot \frac{M_s}{R} + \frac{h_s + 3 \times I_s}{R}\right) \qquad (3.3)$$

Se observa ca, chiar daca rata de miss M_S este foarte aproape de cea a victim cache-ului simplu, sistemele ce folosesc *selective victim cache* pot totusi oferi o îmbunatatire substantiala a performantei superioara sistemelor cu victim cache simplu, din urmatoarele doua motive:

1. Rata de miss locala în cache-ul principal poate fi îmbunatatita printr-un plasament mai bun al blocurilor.
2. Numarul de interschimbari poate descreste ca rezultat al algoritmului de predictie. Aceasta reduce media penalizarii pentru accesele care sunt servite de victime cache, în special când numarul de cicli folositi la o interschimbare este ridicat.

Se foloseste timpul mediu de acces la memorie pentru un sistem cu un cache "2-way associative", ca o referinta pentru evaluarea performantei sistemului cu *selective victim cache*. Pentru estimarea timpului de acces la un cache cache "2-way associative", se presupune ca penalitatea în nanosecunde pentru al doilea nivel al ierarhiei de memorie ramâne aceeasi ca si în cazul cache-ului mapat direct. Pot exista unele constrângeri de implementare care afecteaza penalitatea în caz de miss. Accesarea magistralei sistem, poate implica o secventa de operatii care necesita un numar fix de perioade de tact. Astfel, numarul de cicli necesari pentru deservirea unui miss nu poate descreste proportional cu cresterea perioadei de tact CPU, rezultând într-o penalizare mai mare în cazul cache-ului cache "2-way associative". Timpul mediu de acces la memorie acestui cache este estimat de relatia:

$$T_2 = clk\left(\frac{clk_{2\text{-}way}}{clk} + p \cdot \frac{M_2}{R}\right) \qquad (3.4)$$

Primul termen reprezinta timpul de acces la cache iar al doilea termen este timpul de acces la nivelul urmator de memorie. Comparând aceasta ecuatie cu (3.1), orice îmbunatatire a performantei se datoreaza celui de-al doilea termen, în timp ce primul termen reprezinta câstigul introdus prin asociativitatea cache-ului asupra timpului de acces. Daca îmbunatatirea datorata celui de-al doilea termen nu este adecvata pentru a compensa acest câstig, performanta cache-ului 2-asociativ poate fi inferioara celei a cache-ului mapat direct.

Îmbunatatirea în performanta obtinuta atât prin victim cache cât si prin selective victim cache variaza în functie de trace, depinzând de marimea lor si de numarul de conflicte de acces pe care schema de predictie le elimina. Chiar pentru programe mici, *selective vitim cache* asigura o îmbunatatire semnificativa comparata cu victim cache-ul simplu, când cache-ul nu este

suficient de mare pentru a memora întreg programul. Stiliadis si Varma, în [6], afirma ca cea mai buna îmbunatatire a performantei în termenii ratei de miss, de aproximativ 33%, este obtinuta pentru cache-uri de instructiuni de 8 pâna la 16 Kocteti. Pentru cache-uri mai mari de dimensiuni 64 pâna la 128 Kocteti, majoritatea trace-urilor pot fi usor memorate în cache si miss-urile de conflict reprezinta un mic procent din numarul total de accese cu miss. În aceste cazuri victim cache-ul simplu este capabil sa elimine majoritatea conflictelor, si performanta sa este comparabila cu cea a selective victim cache-ului.

O problema potentiala cu algoritmul de predictie dezvoltat în *selective victim cache* este aceea ca, performanta sa se poate degrada odata cu cresterea dimensiunii blocului, ca rezultat al partajarii bitilor de stare de cuvinte din interiorul aceluiasi bloc. *Selective victim cache* asigura o îmbunatatire semnificativa a ratei de miss indiferent de dimensiunea blocului. Pentru blocuri de dimensiune de 4 octeti, selective victim cache reduce rata de miss cu aproximativ 30% fata de o arhitectura cache cu mapare directa, în timp ce pentru blocuri de dimensiuni de 64 octeti, rata de miss este redusa cu aproape 50%. Aceste rezultate contrazic comportamentul excluziunii dinamice [7], unde reducerea ratei de miss scade cu cresterea dimensiunii blocului. Rezultatele se datoreaza mentinerii la aceeasi dimensiune a victim cache-ului în termenii numarului de blocuri, astfel ca, o crestere a dimensiunii blocului determina o crestere efectiva a capacitatii cache-ului. Aceasta crestere în capacitate compenseaza mai mult decât orice degradare a ratei de predicti prin cresterea dimensiunii blocului. Acest fapt nu creste semnificativ complexitatea implementarii victim cache-ului, deoarece asociativitatea ramâne aceeasi. Indiferent de marimea cache-ului, numarul de interschimbari prin folosirea selective victim cache-ului este redus cu 50% sau mai mult fata de folosirea unui victim cache simplu. Când dimensiunea blocului este mai mare, în functie de implementare, operatia de interschimbare poate necesita câtiva cicli. Prin îmbunatatirea atât a ratei de hit cât si a numarului de interschimbari, selective victim cache-ul poate creste semnificativ performanta primului nivel de cache, superioara victim cache-ului simplu si cache-ului “two-way set associative”. Pentru diverse dimensiuni de cache, îmbunatatirea ratei de miss la cache-ul two-way semiasociativ nu este suficienta pentru a compensa cresterea timpului de acces, rezultând într-o crestere neta a timpului mediu de acces la memorie superior cache-urilor mapate direct. Cea mai mare crestere în performanta a selective victim cache-ului superioara cache-ului semiasociativ, este aproximativ 25%, obtinuta pentru dimensiuni de cache-uri de 16-64 Kocteti.

Politica de scriere implementata este write back cu write allocate. Pentru a mentine proprietatea de incluziune multinivel, blocurile din cache-ul de pe nivelul L1 au fost invalidate când au fost înlocuite pe nivelul L2 de cache. Desi selective victim cache-ul produce îmbunatatiri semnificative ale ratei de hit comparativ cu cache-urile mapate direct de dimensiune redusa, performanta sa este inferioara celei obtinuta folosind victim cache simplu. De fapt, îmbunatatirile ratei de miss variaza semnificativ în functie de trace-uri. Sunt doua motive care explica acest rationament: primul este natura acceselor la memorie a programelor folosite. Programele care implica o alocare statica a datelor si structurilor de date, arata o îmbunatatire cu selective victim cache, ca rezultat al folosirii algoritmului de predictie. Structurile de date principale ale acestor programe sunt vectori. Algoritmul de predictie este capabil sa rezolve un numar mare de conflicte în aceste cazuri, fara sa acceseze al doilea nivel. În programele cu alocare dinamica a memoriei si folosire a extensiei de pointeri, conflictele sunt mai greu de rezolvat de catre algoritmul de predictie. Ratele de miss în situatia folosirii selective victim cache-ului pentru aceste trace-uri sunt mai mari decât în cazul folosirii unui simplu victim cache. În timp ce pentru selective victim cache presupunem un al doilea nivel de cache si mentinem proprietatea de incluziune, simularea victim cache-ului simplu presupune ca al doilea nivel al ierarhiei este memoria principala.

Chiar daca îmbunatatirile asupra ratei de miss sunt mai putin convingatoare în cazul cache-urilor de date comparativ cu cel de instructiuni, selective victim cache poate reduce timpul mediu de acces la memorie pentru primul nivel al cache-ului de date prin reducerea numarului de interschimbari. Pentru cache-uri de dimensiuni pâna la 64 Kocteti, numarul de interschimbari pentru selective victim cache este mult mai mic decât cel pentru victim cache simplu. În câteva cazuri îmbunatatirea este mai mare de 50%. Numarul de interschimbari pentru victim cache simplu descreste sub selective victim cache pentru dimensiuni mai mari sau egale cu 128 Kocteti. Astfel, pentru cache-uri de 128 Kocteti, selective victim cache este capabil sa reduca rata de miss prin cresterea numarului de interschimbari. Desi, victim cache-ul simplu, pare sa se comporte mai bine decât selective victim cache-ul, pentru cache-uri de dimensiuni sub 8 Kocteti, din punct de vedere al timpului mediu de acces. Performanta cache-ului semiasociativ este inferioara ambelor (simplu victim cache si selective victim cache), dar superioara cache-ului mapat direct.

Trace-urile de date sunt caracterizate de o rata de miss mai mare decât trace-urile de instructiuni. În plus miss-urile de conflict sunt raspunzatoare de procentul ridicat din rata totala de miss. Sunt doua consecinte ale acestui fapt: primul, efectul reducerii ratei de miss asupra timpului de acces la

memorie este mai pronuntat, iar al doilea, cache-urile semiasociative pe 2 cai asigura îmbunatatiri în timpul mediu de acces chiar si pentru cache-uri mari, spre deosebire de cache-urile de instructiuni, unde avantajul obtinut prin reducerea ratei de miss datorata cresterii asociativitatii este mai mare decât câstigul obtinut asupra timpului de acces la cache. Concluzionam ca, atât victim cache-ul simplu cât si selective victim cache-ul sunt mult mai putin atractive pentru folosire în cache-ul de date comparativ cu cel de instructiuni.

În continuare se analizeaza modul în care informatiile de stare de care are nevoie schema de predictie dezvoltata în selective victim cache pot fi stocate în interiorul ierarhiei de memorie. Dupa cum s-a aratat schema selective victim cache-ului necesita doi biti de stare pentru a pastra informatii despre istoria blocurilor din cache - bitul sticky si bitul hit. Bitul sticky este asociat logic cu blocul din cache-ul principal. De aceea este normal sa se memoreze acest bit în cache-ul mapat direct ca parte a fiecarui bloc. Pe de alta parte, bitul hit este asociat logic cu fiecare bloc din memoria principala. Astfel, într-o implementare perfecta, bitii de hit trebuie memorati în memoria principala. Aceasta abordare este impracticabila în majoritatea cazurilor.

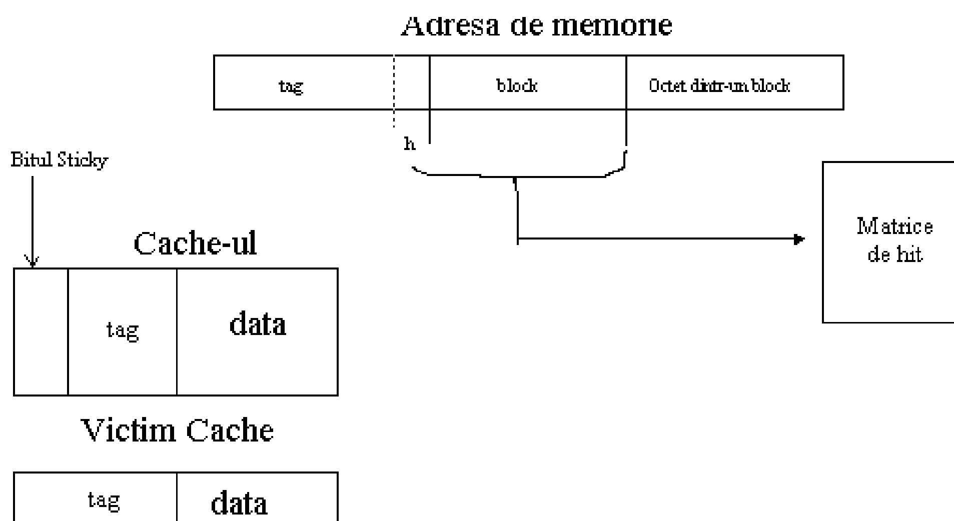


Figura 3.8. Implementarea schemei de memorare a bitilor de hit

Daca ierarhia de memorie include un la doilea nivel de cache, este posibil sa se memoreze bitii de hit în cadrul blocurilor din acest nivel. Când un bloc este adus pe nivelul L1 de cache din nivelul L2, o copie locala a bitului de hit este memorata în blocul de pe nivelul L1. Aceasta elimina nevoia de acces a nivelului L2 de cache de fiecare data când bitul hit este

actualizat de catre algoritmul de predictie. Când blocul este înlocuit din nivelul L1 de cache, bitul hit corespondent este copiat în nivelul L2. O problema ar fi însa aceea ca, multiple locatii din memoria principala sunt fortate sa împarta acelasi bit de pe nivelul L2. Astfel, când un bloc este înlocuit de pe nivelul L2 de cache, toate informatiile lui de stare se pierd, reducând eficacitatea algoritmului de predictie. De fiecare data când un bloc este adus pe nivelul L2 de cache din memoria principala, bitul hit al sau trebuie setat la o valoare initiala. Pentru o secventa specifica de acces, valori initiale diferite pot produce rezultate diferite. Cu cache-urile de pe nivelul L2 de dimensiuni mari, efectul valorilor initiale este probabil mai mic.

O tratare alternativa este de a mentine bitii de hit în interiorul CPU, în cadrul nivelului L1 de cache. În abordarea lui Stiliadis si Varma, un sir de biti de hit numit *hit array* este mentinut ca parte a nivelului L1 de cache. Fiecare bloc de memorie este asociat unuia din bitii acestui sir. Lungimea sirului este inevitabil mai mica decât numarul maxim de blocuri care pot fi adresate. Deci, mai mult de un bloc va fi mapat aceluiasi bit de hit, cauzând datorita interferentelor un aleatorism ce trebuie introdus în procesul de predictie. Desi, aceasta poate potential reduce performanta selective victim cache-ului, rezultatele simularilor nu confirma acest lucru chiar si pentru siruri de hit de dimensiune modesta.

Implementarea nivelului L1 de cache sistem este prezentata în Figura 4. Bitul sticky este mentinut cu fiecare bloc în cache-ul principal. Nici un bit de stare nu este necesar în victim cache. Bitii de hit sunt pastrati în *hit array*, adresati de o parte a adresei de memorie. Dimensiunea sirului de hit bit este aleasa ca un multiplu al numarului de blocuri din cache-ul principal. Astfel,

Dimensiunea sirului hit array = Numar de blocuri în cache-ul principal ´ H

unde H determina gradul de partajare a bitilor de hit de catre blocurile memoriei principale. Se presupune ca H este o putere a lui 2, $H=2^h$. *Hit array* poate fi adresat de adresa de bloc concatenata cu cei mai putin semnificativi biti *h*, din partea de tag a adresei. O valoare mare pentru H implica mai putine interferente între blocurile conflictuale la bitii de hit. Daca H este ales ca raport dintre dimensiunea cache-ului de pe nivelul L2 si cea a cache-ului de pe nivelul L1 (principal), atunci efectul este similar cu mentinerea bitilor hit în nivelul L2 de cache.

O problema a implementarii schemei atât a victim cache-ului cât si a selective victim cache-ului este costul implementarii victim cache-ului full asociativ. Chiar si atunci când aceste cache-uri sunt foarte mici, costul hardware al memoriei adresabila contextual (CAM) poate fi semnificativ.

Cache-urile complet asociative cu algoritm de înlocuire LRU pot uneori suferi de o rata de miss mai ridicata decât cache-urile two-way asociative deoarece algoritmul de înlocuire nu este cel optimal [6, 10]. Efectul ambelor probleme de mai sus poate fi diminuat prin reducerea asociativitatii victim cache-ului. Cu un victim cache semiasociativ pe 2 cai, nu se observa nici o crestere a ratei de miss la nivelul urmator al ierarhiei de memorie pentru nici o instructiune din trace-urile simulate, atât în victim cache simplu cât si în selective victim cache. Surprinzator, victim cache-ul semiasociativ pe 2 cai poate îmbunatati rata de miss si timpul mediu de acces pentru mai multe trace-uri. Acest comportament se datoreaza algoritmului de înlocuire LRU dezvoltat în victim cache-ul complet asociativ. Blocurile mutate în victim cache-ul complet asociativ ca rezultat al conflictelor din cache-ul principal sunt înlocuite frecvent înainte de a fi accesate din nou. Victim cache-ul semiasociativ pe 2 cai asigura o mai buna izolare pentru blocurile sale în multe cazuri, micsorând rata de miss în victim cache. În plus, datorita dimensiunii sale reduse, miss-urile de conflict formeaza doar o mica fractiune din numarul total de accese cu miss în victim cache comparativ cu miss-urile de capacitate. Aceasta limiteaza îmbunatatirea ratei de miss prin cresterea asociativitatii victim cache-ului, chiar cu un algoritm optimal de înlocuire. Se observa ca, victim cache-ul full asociativ poate îmbunatati dramatic rata de miss în cazul conflictelor ce implica mai mult de trei blocuri, blocurile conflictuale fiind retinute în victim cache între accese.

Cu un victim cache simplu continutul cache-ului principal mapat direct este neafectat de asociativitatea acestuia. Astfel, rata de miss locala ramâne neschimbata în timp ce se variaza asociativitatea victim cache-ului. Prin urmare toate îmbunatatirile efectuate asupra ratei de miss la nivelul L1 de cache pot fi atribuite îmbunatatirii ratei de miss locale a victim cache-ului. Cu victim cache-ul selectiv, asociativitatea poate afecta potential atât rata de miss locala a cache-ului principal cât si numarul de interschimbari dintre cele doua cache-uri. Comparatia timpului de acces tine cont de schimbarile aparute în rata de miss si numarul de interschimbari (substantial micsorat) si de aceea timpul mediu de acces reprezinta o masura mai buna pentru caracterizarea efectului de asociativitate al victim cache-ului asupra performantei sistemului. Chiar cu un victim cache mapat direct, timpul mediu de acces este mai mare sau egal decât cel din cazul victim cache-ului complet asociativ. Când folosim un victim cache de date semiasociativ pe 2 cai, rezultatele sunt mai proaste decât acelea cu un victim cache complet asociativ, atât pentru victim cache simplu cât si pentru victim cache-ul selectiv. Acest lucru nu surprinde, dând conflictelor de acces la date o natura aleatorie. Astfel, un cache complet asociativ poate fi înca atractiv

când este folosit ca si cache de date. Totusi, îmbunatatirile observate sunt mai mici.

Chiar daca nu este nici o îmbunatatire a ratei de hit în cache-ul principal, schema victim cache-ului selectiv poate totusi asigura o îmbunatatire a performantei superioara victim cache-ului simplu. Pentru schema cu SVC rezultatele demonstreaza ca îmbunatatirile de performanta sunt puternic determinate de impactul algoritmului de predictie asupra numarului de interschimbari cu cache-ul mapat direct. Algoritmul poate de asemenea contribui la o mai buna plasare a blocurilor în cache, reducând numarul de accese în victim cache si generând rate de hit ridicate în cache-ul mapat direct.

Folosirea victim cache-ului selectiv determina îmbunatatiri ale ratei de miss precum si ale timpului mediu de acces la memorie, atât pentru cache-uri mici cât si pentru cele mari (4Kocteti - 128 Kocteti). Simulari facute pe trace-uri de instructiuni a 10 benchmark-uri SPEC'92 arata o îmbunatatire de aproximativ 21% a ratei de miss, superioara folosirii unui victim cache simplu de 16 Kocteti cu blocuri de dimensiuni de 32 octeti; numarul blocurilor interschimbate între cache-ul principal si victim cache s-a redus cu aproximativ 70%.

3.2. MEMORIA VIRTUALA

Memoria virtuala reprezinta o tehnica de organizare a memoriei prin intermediul careia programatorul "vede" un spatiu virtual de adresare foarte mare si care, fara ca programatorul sa "simta", este mapat în memoria fizic disponibila. Uzual, spatiul virtual de adrese corespunde suportului disc magnetic, programatorul având iluzia prin mecanismele de memorie virtuala (MV), ca detine o memorie (virtuala) de capacitatea hard-discului si nu de capacitatea memoriei fizice preponderenta DRAM (limitata la 64 ÷1024 Mo la ora actuala).

În cazul MV, memoria principala este analoaga memoriei cache între CPU (*Central Processing Unit*) si memoria principala, numai ca de aceasta data ea se situeaza între CPU si discul hard. Deci memoria principala (MP) se comporta oarecum ca un cache între CPU si discul hard. Prin mecanismele de MV se mareste probabilitatea ca informatia ce se doreste a fi accesata de catre CPU din spatiul virtual (disc), sa se afle în MP, reducânduse astfel dramatic timpul de acces de la 8 ÷ 15 ms la 45 ÷ 70 ns în

tehnologiile actuale (1999) ! De obicei, spatiul virtual de adresare este împartit în entitati de capacitate fixa (4 ÷ 64 Ko actualmente), numite pagini. O pagina poate fi mapata în MP sau pe disc.

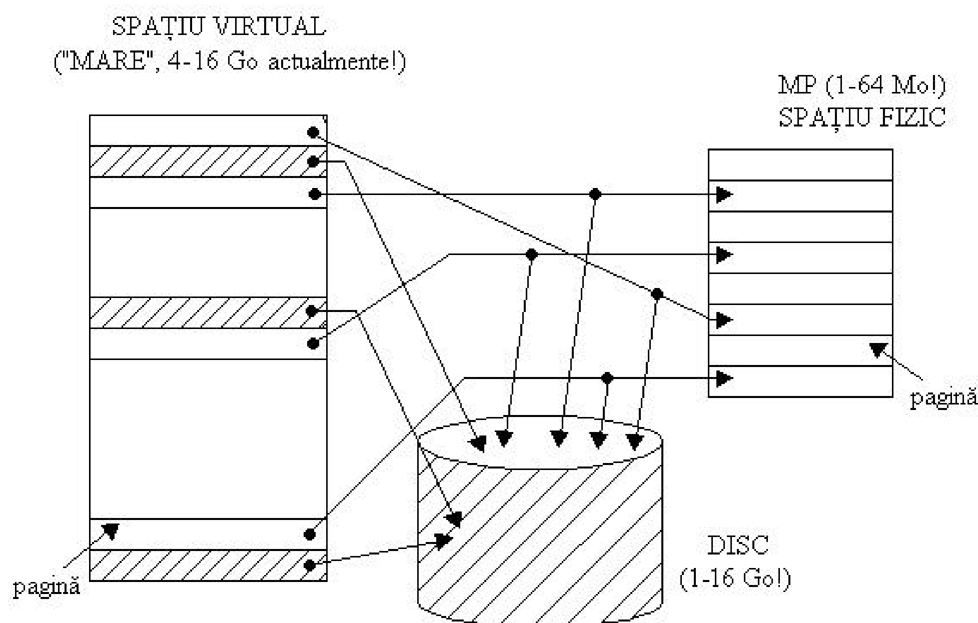


Figura 3.9. Maparea adreselor virtuale în adrese fizice

În general, prin mecanismele de MV, MP contine paginile cel mai recent accesate de catre un program, ea fiind dupa cum am mai aratat, pe post de "cache" între CPU si discul hard. Transformarea adresei virtuale emisa de catre CPU întro adresa fizica (existenta în spatiul MP) se numeste mapare sau translatare. Asadar mecanismul de MV ofera o functie de relocare a programelor (adreselor de program), pentru ca adresele virtuale utilizate de un program sunt relocate spre adrese fizice diferite, înainte ca ele sa fie folosite pentru accesarea memoriei. Aceasta mapare permite aceluiasi program sa fie încarcat si sa ruleze oriunde ar fi încarcat în MP, modificarile de adrese realizându-se automat prin mapare (fara MV un program depinde de obicei în executia sa de adresa de memorie unde este încarcat).

MV este un concept deosebit de util în special în cadrul sistemelor de calcul multiprogramate care - de exemplu prin "time-sharing" - permit executia cvasi-simultana a mai multor programe (vezi sistemul de operare

WINDOWS 2000, NT, Unix, Ultrix etc.). Fiecare dintre aceste programe are propriul sau spatiu virtual de cod si date (având alocate un numar de pagini virtuale), dar în realitate toate aceste programe vor partaja aceeasi MP, care va contine dinamic, pagini aferente diverselor procese. Paginile vor fi dinamic încarcate de pe disc în MP respectiv evacuate din MP pe disc (spre a permite încarcarea altora, mai "proaspete").

Când o pagina accesata nu se gaseste în MP, ea va trebui adusa prin declansarea unui mecanism de exceptie, de pe disc. Acest proces – analogul miss-urilor de la cache-uri – se numeste "*page fault*" (PF). Evenimentul PF va declansa un mecanism de exceptie care va determina intrarea într-o subrutina de tratare a evenimentului PF. Aici – prin software deci – se va aduce de pe disc în MP pagina dorita dupa ce, fireste, în prealabil s-a evacuat eventual o alta pagina din MP. Acest proces este unul de lunga durata, necesitând câteva ms bune la ora actuala. Având în vedere multitaskingul, MV trebuie sa asigure si mecanismul de protectie a programelor (ex. sa nu permita unui program utilizator sa scrie zona de date sau cod a sistemului de operare sau a altui program, sa nu permita scrierea într-o pagina accesabila numai prin citire etc.).

În implementarea MV trebuie avute în vedere urmatoarele aspecte importante:

- ♦ paginile sa fie suficient de mari (4 ko ÷16 ko … 64 ko) astfel încât sa compenseze timpul mare de acces la disc (9 ÷12 ms).
- ♦ organizari care sa reduca rata de evenimente PF, rezultând un plasament flexibil al paginilor în memorie (MP)
- ♦ PF-urile trebuie tratate prin software si nu prin hardware (spre deosebire de miss-urile în cache-uri), timpul de acces al discurilor permitând lejer acest lucru.
- ♦ scrierile în MV se fac dupa algoritmi tip "Write Back" si nu "Write Through" (ar consuma timp enorm).

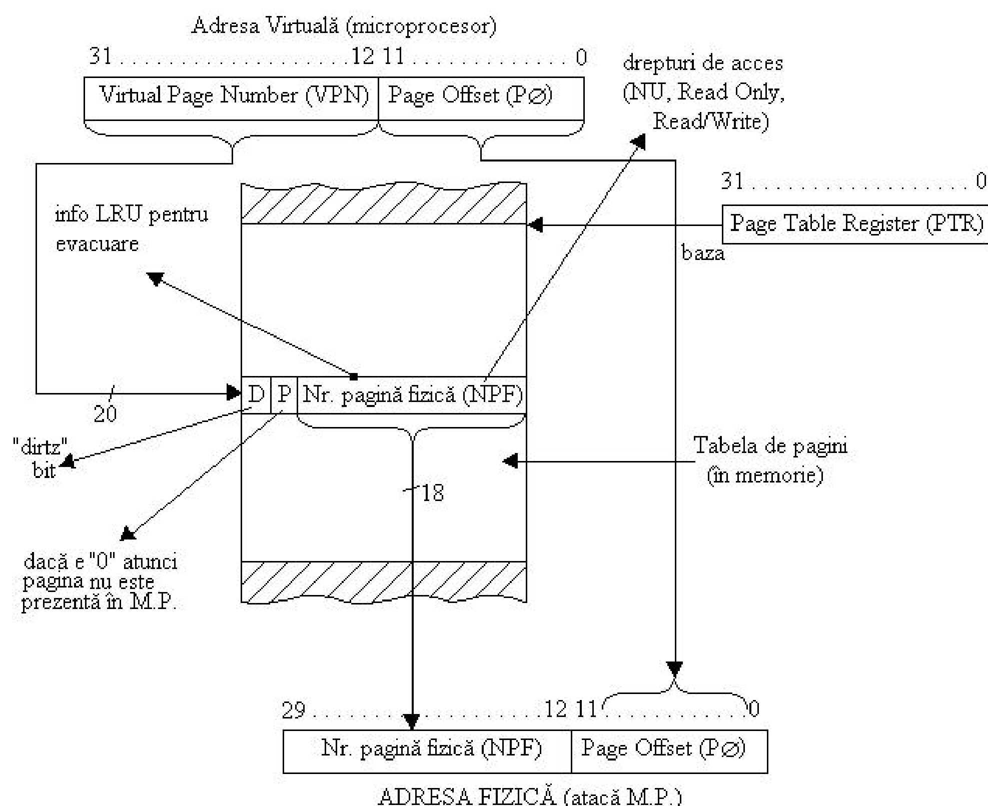


Figura 3.10. Translatare adresa virtuala în adresa fizica

Obs. Fiecare program are propria sa tabela de pagini care mapeaza spatiul virtual de adresare al programului într-un spatiu fizic, situat în M.P.

Tabela de pagini + PC + registrele microprocesorului formeaza starea unui anumit program. Programul + starea asociata caracterizeaza un anumit proces (task). Un proces executat curent este activ, altfel el este inactiv. Comutarea de taskuri implica inactivarea procesului curent si activarea altui proces, inactiv pâna acum rezultând deci ca fiind necesara salvarea/restaurarea starii proceselor. Desigur, sistemul de operare (S.∅.) trebuie doar sa reâncarce registrul pointer al adresei de baza a paginii (PTR) pentru a pointa la tabela de pagini aferenta noului proces activ.

Exceptia Page Fault (P.F.)

Apare în cursul mecanismului de translatare a adresei virtuale în adresa fizica, daca bitul P = 0. Ca urmare, printr-o procedura de exceptie se

da controlul unui handler al S.∅. în vederea tratarii. Aici S.∅. va trebui sa decida ce pagina din M.P. va trebui evacuata în vederea încarcarii noii pagini de pe disc. În general, ca principiu, se poate merge pe ideea LRU ("*Least Recently Used*"), adica va fi evacuata pagina care nu a mai fost accesata de catre CPU de cel mai mult timp (se merge deci implicit pe principiul localitatii temporale).

Exemplu: CPU a accesat în ordine paginile: 10,12,9,7,11,10 iar acum acceseaza pagina 8 care nu este prezenta în MP ⇒ evacueaza pagina 12 ! Daca urmatorul acces genereaza PF ⇒ evacueaza pagina 9, s.a.m.d.

Obs. Unele masini (ex. Pentium) implementeaza în tabela de pagini câte un bit de referinta pentru fiecare pagina. Acest bit este setat la fiecare accesare a acelei pagini. S.∅. sterge periodic acesti biti – nu înainte de a le memora starea – astfel încât sa implementeze pentru fiecare pagina un contor; astfel, bazat pe starea de moment a acestor contoare, se stabileste care pagina va fi evacuata.

Scrierile în MP se desfasoara dupa urmatoarele principii:

- strategie similara cu cea de tip write-back de la memoriile cache (*copy-back*)
- se adauga un "Dirty Bit" (D) în tabela de pagini pentru fiecare pagina. Bitul D e setat la fiecare scriere în pagina ⇒ la evacuare, o pagina având bitul D=0, nu are rost sa se evacueze efectiv pe disc ⇒ pierdere mare de timp ⇒ minimizare scrieri !

Translation – Lookaside Buffers (TLB)

Prin paginare, fiecare acces la o data necesita 2 accese la memorie: unul pentru obtinerea adresei fizice din tabela de pagini, iar celalalt pentru accesarea propriu-zisa a datei în M.P. În vederea reducerii acestui timp de acces (dublu), tabela de pagini este "casata" (memorata partial) în CPU. Memoria cache care memoreaza maparea tabelei de pagini se numeste TLB (Translation Lookaside Buffer). Ca orice cache, TLB-ul poate avea diferite grade de asociativitate. Exista evacuari/încarcari între TLB si tabela de pagini din M.P.

Deoarece TLB-ul este implementat în general "on-chip", capacitatea sa este relativ mica (32÷1024 intrari), în comparare cu tabela de pagini care are 1 ÷ 4 M intrari. De obicei TLB-ul se implementeaza complet asociativ (full-associative), pentru a avea o rata de miss scazuta (0,01 % ÷ 0,1 % ÷ 1

%). Missurile în TLB se pot rezolva atît prin protocol hardware cât si printr-un handler software.

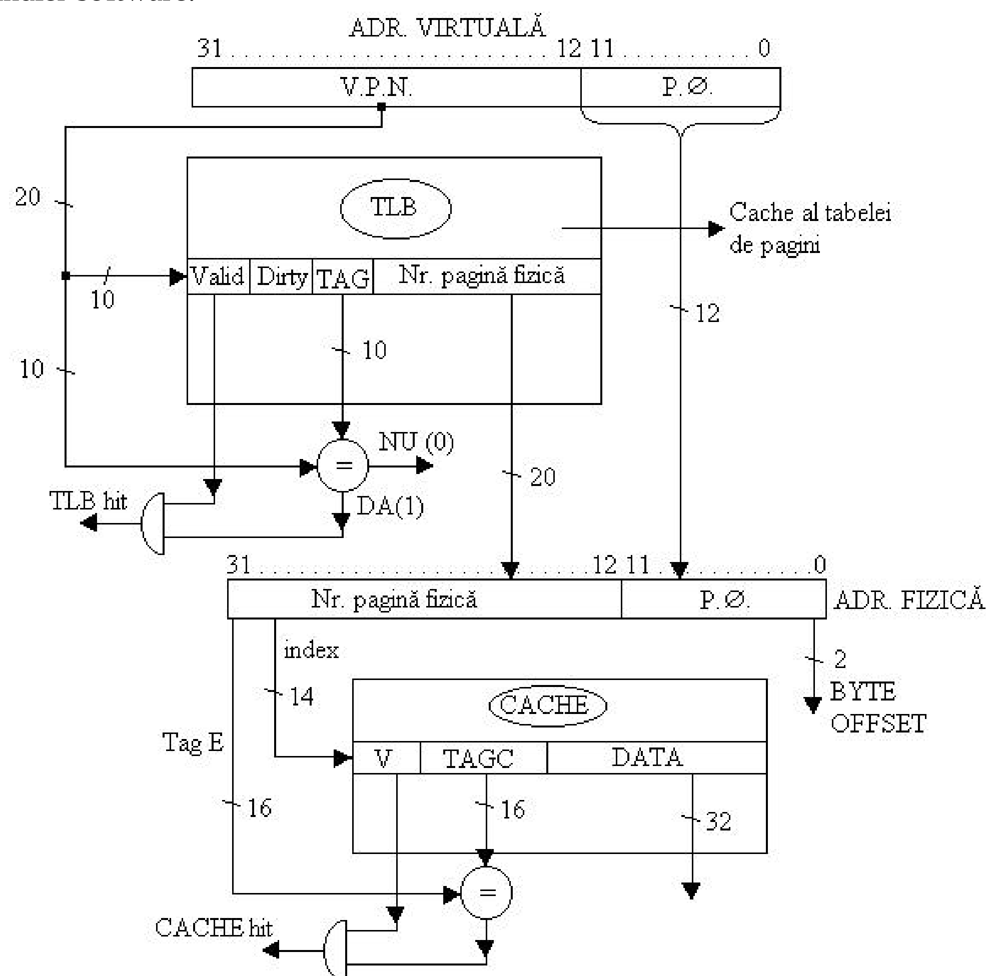


Figura 3.11. Relatia TLB - cache într-un microsistem DEC 3100 (microprocesor MIPS-R2000)

Obs. Ar fi mai rapid daca s-ar adresa cache-ul cu adresa virtuala (cache-uri virtuale) si nu cu cea fizica. Probleme/solutii în acest sens sunt date în [8] (comutari taskuri- Process Identifier, antialias, "page colouring" etc.) O solutie simpla si imediata ar consta în adresarea cache-ului cu bitii P∅ care sunt nemodificati prin procesul de translatare. Desigur în acest caz este necesar ca dimensiunea cache ≤ dimensiunea paginii.

Protectia în sistemele cu M.V.

Desi fiecare proces are propriul sau spatiu virtual de adresare, memoria fizica (MP) este partajata între mai multe procese (procese utilizator, S∅, driverele I/O etc.). Desigur, trebuie sa se controleze strict accesul unui proces în zonele de cod si date ale altui proces rezultând necesitatea protectiei la scrieri/citiri. De exemplu, numai S.∅. trebuie sa poata modifica tabelele de pagini aferente diferitelor procese. În vederea implementarii protectiilor, hardul trebuie sa asigure cel putin urmatoarele 3 conditii:

1. Cel putin 2 moduri distincte de rulare a unui program:modul supervizor (kernel, executiv) în care un proces poate sa execute orice instructiuni si sa acceseze oricare resurse si respectiv modul user în care un proces are o multime de restrictii legate de protectia si securitatea sistemului.
2. Sa existe o parte a starii CPU în care un proces user sa nu poata scrie. De exemplu: biti de stare user/kernel, registrul PTR, bitul validare/invalidare, exceptii, pagini kernel (ACCES) etc. Posibilitatea unui proces user sa scrie astfel de resurse ar determina S.∅. (proces supervizor) sa nu poata controla procesele user.
3. Mecanismele de tranzitie a procesorului din modul supervizor în modul user si invers. Tranzitia user-supervizor în modul user se poate face printr-o exceptie (întrerupere) sau printr-o instructiune speciala de tip SYSTEM CALL, care transfera controlul la o adresa dedicata din spatiul de cod supervizor (CALL GATE – la Pentium). Se salveaza PC-ul si contextul procesului curent si CPU e plasat în modul de lucru anterior (user aici).

De asemenea, din modul “supervizor” se poate trece în modul “user” prin simpla modificare a bitilor de mod (e permis !). De exemplu, sa presupunem ca un proces P2 doreste sa îi transmita (citire) anumite date, printr-o pagina proprie, unui alt proces P1. Pentru asta, P2 ar putea apela o rutina a S.∅. (printr-un SYSTEM CALL), care la rîndul ei (fiind privilegiata!) va crea o intrare în tabela de pagini a lui P1 care sa se mapeze pe pagina fizica pe care P2 doreste s-o puna la dispozitie. S.∅. (supervizor) poate sa utilizeze bitul “Write Protection” pentru a împiedica procesul P1 sa altereze respectiva pagina. Si alti biti de control de tip “drepturi de acces" în pagina pot fi inclusi în tabela de pagini si în TLB.

Obs. În cazul unei comutari de taskuri de la procesul P1 la procesul P2, TLB-ul trebuie golit din 2 motive: în caz de hit P2 sa nu utilizeze paginile lui P1 si respectiv sa se încarce în TLB intrarile din tabela de

pagini a procesului P2 (pointata de noul PTR). Asta se întâmpla numai daca P1 si P2 folosesc VPN-uri identice (bitii 31 ÷ 12 din adrese virtuala). Pentru a nu goli TLB-ul prea des, se prefera adaugarea la tag-ul acestuia a unui câmp numit PID (“Process Identifier” – identificator al procesului), care va contribui corepunzator la HIT. Aceasta informatie (PID) este tinuta de obicei într-un registru special, ce va fi încarcat de catre S.∅. la fiecare comutare de taskuri. Ca si consecinta se evita în majoritatea cazurilor golirea (si implicit reumplerea!) TLB-ului.

În concluzie, foarte succint, protectia este asigurata în principal prin:
- moduri de lucru CPU de diferite nivele de privilegiu
- control strict al S.∅. asupra tranzitiilor din user în kernel (prin CALL GATES-uri - porti de apel - la o anumita adresa determinata din spatiul de cod kernel)
- protectie a paginilor prin “drepturi de acces” la pagina (read only, read/write etc).

Tratarea miss-urilor în TLB si a PF-urilor

Pe durata procesului de translatare a adresei virtuale în adresa fizica pot sa apara 2 evenimente de exceptie:

1. TLB miss, dar pagina accesata este prezenta în memoria fizica (M.P.)
2. Page Fault (PF), adica TLB miss urmat de faptul ca pagina dorita nu este prezenta în tabela de pagini rezidenta în M.P. (bit P=0).

Un TLB miss genereaza de obicei o procedura hardware de aducere a numarului paginii fizice din tabela de pagini. Aceasta operatie se poate implementa prin hardware, ea durând un timp relativ scurt (cca. 20 – 50 tacte CPU).

Tratarea PF în schimb, necesita un mecanism de tratare al exceptiei care sa întrerupa procesul activ, sa transfere controlul rutinei de tratare PF (S.∅.) si apoi sa redea controlul procesului întrerupt. PF va fi recunoscut pe parcursul ciclilor de acces la memorie. Cum instructiunea care a cauzat PF trebuie reluata, rezulta ca trebuie salvat în stiva (automat) PC-ul aferent acesteia. Pentru asta, exista un registru special EPC (Exception PC), întrucât PC-ul propriu-zis poate sa fie mult incrementat sau chiar complet altul (din motive de prefetch, branch-uri etc.). Mai apoi, printr-un sistem de întreruperi (vectorizate) se da controlul rutinei de tratare din cadrul S.∅. Aici, se afla cauza exceptiei prin consultarea registrului “cauza exceptie” iar apoi se salveaza întreaga stare (context) a procesului întrerupt (registrii

generali, PTR, EPC, registri “cauza exceptie“ etc.). Daca PF-ul a fost cauzat de un “fetch sau write data”, adresa virtuala care a cauzat PF trebuie calculata din însasi formatul instructiunii pe care PF s-a produs (PC-ul aferent acesteia e memorat în EPC), de genul “base + offset”.

Odata stiuta adresa virtuala care a cauzat PF, rutina de tratare a S.∅. aduce pagina de pe disc în MP, dupa ce mai întâi a evacuat (LRU) o pagina din MP pe disc. Cum accesul pe disc dureaza mii de tacte, uzual S.∅. va activa un alt proces pe aceasta perioada.

Segmentarea

Constituie o alta varianta de implementare a MV, care utilizeaza în locul paginilor de lungime fixa, entitati de lungime variabila zise segmente. În segmentare, adresa virtuala este constituita din 2 cuvinte: o baza a segmentului si respectiv un deplasament (offset) în cadrul segmentului. Datorita lungimii variabile a segmentului (de ex. 1 octet ÷ 2^{32} octeti la arhitecturile Intel Pentium), trebuie facuta si o verificare a faptului ca adresa virtuala rezultata (baza + offset) se încadreaza în lungimea adoptata a segmentului. Desigur, segmentarea ofera posibilitati de protectie puternice si sofisticate a segmentelor. Pe de alta parte, segmentarea induce si numeroase dezavantaje precum:

- 2 cuvinte pentru o adresa virtuala, necesare având în vedere lungimea variabila a segmentului. Asta complica sarcina compilatoarelor si a programelor
- încarcarea segmentelor variabile în memorie mai dificila decât la paginare
- fragmentare a memoriei principale (portiuni nefolosite)
- frecvent, trafic ineficient MP-disc (de exemplu pentru segmente “mici”, transferul cu discul e complet ineficient – accese la nivel de sector = 512 octeti)

Exista în practica si implementari hibride segmentare – paginare.

4. O MICROARHITECTURA MODERNA REPREZENTATIVA: HSA

4.1. INTRODUCERE

Caracteristica principala a arhitecturii HSA (Hatfield Superscalar Architecture) o constituie exploatarea paralelismului la nivelul instructiunii, într-un mediu superscalar, prin schedulling agresiv aplicat codului sursa în momentul compilarii. HSA reprezinta un procesor puternic paralel, caracterizat de un set de instructiuni RISC simple ce permit exploatarea eficienta a unitatilor functionale pipelineizate. Instructiunile sunt întâi extrase din cache-ul de instructiuni sau memoria principala (în caz de miss în cache) si stocate în bufferul de prefetch. La nivelul acestuia, dupa verificarea dependentelor de date între instructiuni si a constrângerilor legate de resurse, grupuri formate de instructiuni sunt expediate spre unitatile functionale de executie. Nu este necesara aplicarea tehnicii de renumire a registrilor, schedullerul CGS tratând probleme legate de renumirea si reordonarea codului. Întrucât arhitectura HSA nu implementeaza mecanismul de branch prediction , schedullerul rezolva problema penalitatilor introduse de instructiunile de salt prin rearanjarea codului în zone (sectiuni) "*branch delay slot*" minimizând întârzierile provocate astfel. Instructiunile sunt trimise spre executie în ordinea lor initiala "*in order*" dar executia lor se poate încheia într-o ordine arbitarara "*out of order*". Unitatile functionale genereaza noile rezultate, le depune pe magistrala rezultat pentru a fi înscrise în file-ul de registre sau înaintate celorlalte unitati functionale care au nevoie de respectiva data. Arhitectura HSA implementeaza trei file-uri de registre pentru memorarea variabilelor întregi, booleene sau flotante.

Executia conditionata este implementata prin atasarea variabilelor garda booleene instructiunilor, permitând procesorului sa distinga între instructiunile apartinând diferitelor fire de executie dar care au fost reorganizate (împachetate) în aceeasi fereastra de întârziere a branch-urilor.

Evacuarea codului executabil conditionat din bufferul de prefetch se face de îndata ce se cunoaste daca instructiunile se vor executa sau nu, prin intermediul variabilelor garda booleene. Evacuarea vremelnica a codului nedorit determina cresterea performantei globale a procesoarelor cu resurse limitate.

Arhitectura superscalara are patru nivele distincte în procesarea instructiunilor. În nivelul **IF** (fetch instructiune) - se calculeaza adresa grupului de instructiuni ce trebuiesc citite din cache-ul de instructiuni sau din memoria principala; dupa citire, blocul de instructiuni este plasat în partea superioara a Buffer-ului de Prefetch. Instructiunile din partea inferioara a buffer-ului sunt selectate si trimise celui de-al doilea nivel: **ID** (decodificare instructiune) - în care sunt decodificate instructiunile aduse, se citesc operanzii din setul de registrii generali, se calculeaza adresa de salt (pentru instructiunile de ramificatie) si respectiv se calculeaza adresa de acces la memorie (pentru instructiunile LOAD sau STORE). Instructiunile sunt apoi pasate unitatilor functionale potrivite, care folosesc operanzii sursa în timpul celei de-a treia faze de procesare a pipe-ului: **ALU/MEM**. În aceasta faza se executa operatia ALU asupra operanzilor selectati în cazul instructiunilor aritmetico-logice si se acceseaza memoria cache de date sau memoria principala (în caz de miss în cache), pentru instructiunile cu referire la memorie (citire sau scriere). Unitatile de executie a instructiunilor LOAD si STORE sunt singurele unitati functionale care interactioneaza în mod direct cu cache-ul de date. În final, avem cel de-al patrulea nivel **WB** (scriere date) - în care, unitatile functionale preiau rezultatul final al instructiunilor aritmetico-logice sau data citita din memorie, si o depun pe magistrala rezultat, de unde este copiata în registrul destinatie din setul de registrii generali.

Arhitectura HSA este puternic parametrizabila. Dimensiunea blocului accesat din cache –ul de instructiuni si capacitatea maxima a bufferului de prefetch sunt parametri care variaza între implementari diferite ale procesorului. Bufferul de prefetch este implementat ca o structura de date dinamica de tip coada ce lucreaza dupa principiul FIFO. Numarul de instructiuni ce pot fi trimise simultan spre executie este dat de numarul “pipe”-urilor logice asigurate de arhitectura superscalara. Într-un ciclu de executie, instructiuni noi din partea inferioara (bottom) a bufferului de prefetch sunt selectate si formeaza grupuri independente gata de executie, fiecarei instructiuni fiindu-i asignat câte un pipe logic. Pipe-urile functioneaza ca dispozitive de rutare a instructiunilor spre unitatile functionale de executie corespunzatoare.

În continuare, se prezinta schema bloc a arhitecturii superscalare HSA, urmând a se discuta despre elementele sale componente pe parcursul acestui capitol.

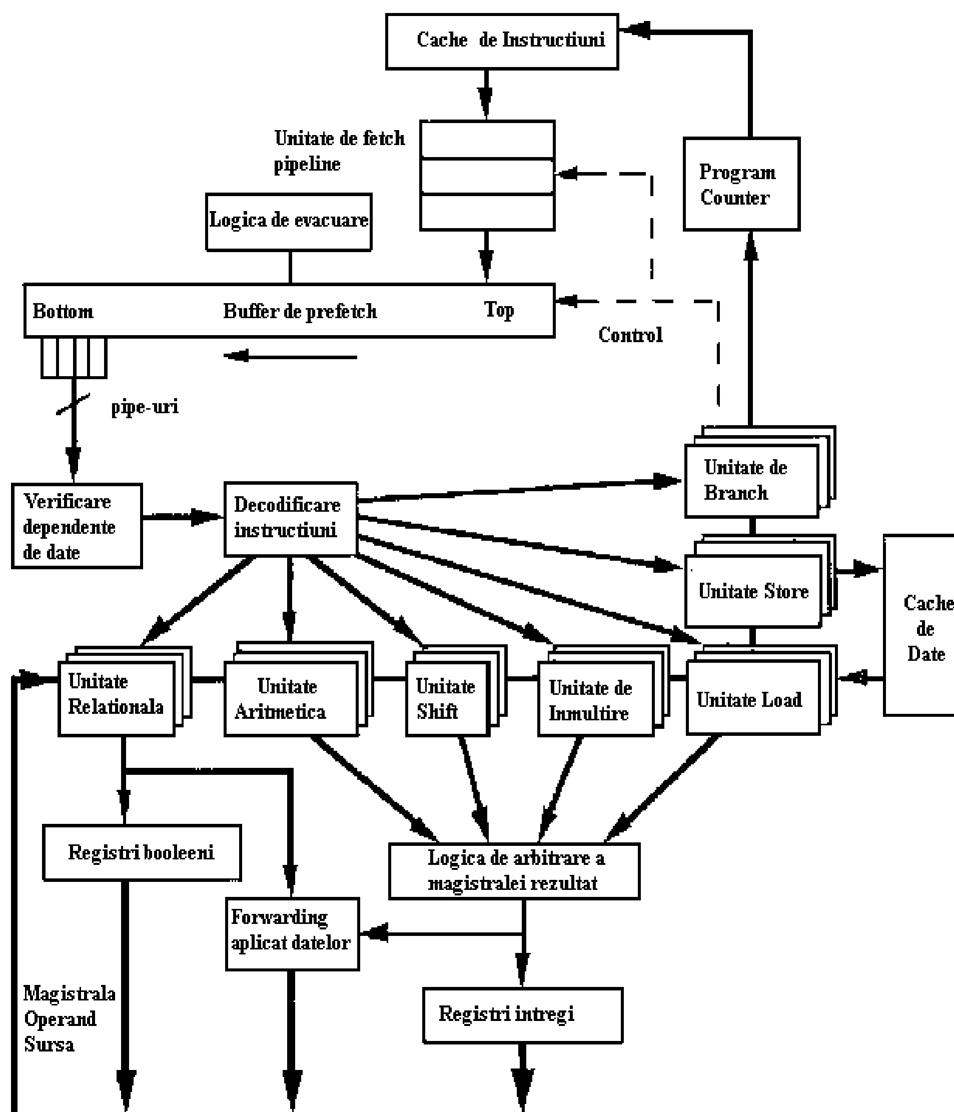


Figura 4.1. Schema bloc a arhitecturii HSA

4.2. ARHITECTURA HSA. COMPONENTE PRINCIPALE.

Modurile de adresare

Pentru instructiunile cu referire la memorie, arhitectura HSA ofera doua moduri de adresare: **indirect registru** si **indexat** (indirect registru + offset). Calculul adresei se realizeaza însumând fie continutul a doi registri fie un registru cu o valoare imediata. Registrul R0 este cablat la masa, ca în toate procesoarele RISC, si este folosit pentru simularea adresarii directe (offset + (R0)).

Daca în cazul unei instructiuni LOAD, adresa de memorie este calculata abia în nivelul EX de procesare, va exista o penalitate de cel putin o perioada de tact în obtinerea valorii dorite din memorie. Aceasta întârziere se rasfrânge asupra executiei instructiunilor succesoare care au ca operand sursa respectiva valoare (cea din memorie), indisponibila înca. Evitarea penalitatii introduse de LOAD se face prin modul de adresare "**OR**". Acest mod substituie, atunci când este posibil (când ultimii *n* biti ai registrului de adresa folosit, unde *n* reprezinta numarul de biti pe care este reprezentat offsetul, sunt zero) operatia de adunare dintre registru si offset (consumatoare de timp prin propagarea transportului) cu cea de SAU logic mult mai rapida. Pretul utilizarii acestui mod de adresare consta în faptul ca structurile de date si memoria stiva trebuie aliniate de catre compilator în zone de memorie de capacitate puteri ale lui 2, ceea ce determina cresterea necesarului de memorie a unui program.

File-urile de registre

Arhitectura HSA defineste trei seturi distincte de registri: întregi, booleeni si flotanti. Numarul registrilor aferenti fiecarui set variaza în functie de fiecare instanta simulata (implementata) însa trebuie sa fie suficient sa satisfaca necesarul tehnicii de schedulling CGS (conditional group schedulling). Tipic, exista 32÷64 registri întregi, 8÷16 registri booleeni si un numar parametrizabil de registri flotanti.

Fiecare registru contine un flag care indica daca data continuta este "*invalida*". Aceasta caracteristica serveste la doua scopuri: executia speculativa instructiunilor si depanarea run-time a programelor de test. Un alt flag poate fi folosit pentru a marca faptul ca un registru este "*indisponibil*", datorita calculari valorii sale de catre unitatile functionale de executie. Instructiunile urmatoare, care au nevoie de respectiva data ca si

operand sursa sunt fortate sa ramâna în stare "*wait*" pâna la generarea rezultatului de catre unitatile functionale.

În general, instructiunile cu operanzi întregi au doi registri sursa si unul destinatie. Instructiunile booleene sunt caracterizate de doi registri sursa si unul sau doi registri destinatie. Folosirea a doi registri destinatie într-o singura instructiune booleana este exploatata prin tehnica CGS, când codul este reorganizat în zone de umplere a "*branch delay slot*" – ului generat de instructiunile de salt în cadrul buclelor de program.

Toate unitatile functionale de executie folosesc magistrala rezultat pentru a scrie valoarea calculata în urma operatiilor efectuate si pentru a înainta noua valoare rezultata altor unitati functionale care o necesita. În functie de variantele de implementare a procesorului, numarul magistralelor rezultat poate fi mai mic decât numarul unitatilor functionale de executie. În aceste cazuri, unitatile functionale trebuie sa partajeze folosirea magistralei rezultat, fiind necesar un mecanism de arbitrare al acesteia care sa decida asupra unitatii care poate înscrie rezultatul pe magistrala. Prioritatea maxima se acorda instructiunilor cu latente mari si acelor instructiuni care se afla deja în stare "*wait*" datorita unei cereri anterioare de alocare a magistralei rezultat, nesatisfacute.

Instructiunile pot fi retinute în bufferul de prefetch datorita dependentelor de tip WAW ("*write after write*"). Într-un astfel de hazard, o instructiune calculeaza un nou rezultat pentru un registru care este folosit ca registru destinatie de catre o alta instructiune, considerata deja expediata spre executie. Daca instructiunea din urma se executa integral înainte ca prima instructiune sa se încheie, este posibil ca valoarea finala memorata în registrul destinatie sa fie incorecta. Totusi, daca o instructiune este promovata speculativ înaintea unui salt conditional de pe o ramura, registrul destinatie nu mai este de nici un folos daca programul urmeaza executia de pe cealalta (alternativa) ramura.

La apelul procedurilor si la folosirea modului de adresare "OR" un numar de 4 registri sunt rezervati compilatorului [16]. Doi registri pointeri la memorie, GP (global pointer) si SP (stack pointer) sunt folositi pentru adresa de baza a obiectelor globale de date, respectiv a segmentului de stiva. Registrul de stare program (SR) poate fi folosit pentru a retine continutul file-ului de registre boolean si a-l salva în stiva în cazul apelurilor de proceduri. Uneori, este folosit si un registru pointer de stiva suplimentar (SP1), atunci când e necesara valoarea anterioara a registrului SP. Doar registrul SR necesita suport arhitectural din partea hardware-ului, celelalte trei registre reprezentând conventii de nume folosite de compilator.

Unitatea de fetch si Bufferul de prefetch

O caracteristica importanta a arhitecturii HSA consta în faptul ca, rata de fetch a instructiunilor (FR – numarul de instructiuni citite simultan din cache-ul de instructiuni sau memoria centrala) nu depinde de numarul instructiunilor trimise simultan spre executie. În fiecare ciclu de executie, grupuri de FR instructiuni sunt extrase din cache-ul de instructiuni sau memorie, de la adresa data de registrul PC (program counter) si, daca exista spatiu disponibil, depuse în bufferul de prefetch. Concomitent, alte grupuri de instructiuni, existente în bufferul de prefetch, sunt selectate pentru trimiterea în executie, instructiunile ramase fiind deplasate în buffer pentru a elibera spatiul, necesar aducerii altor FR instructiuni din cache în ciclul urmator de executie. Desi dependentele reale de date – RAW ("*read after write*") limiteaza rata de procesare, aceasta nu are efect direct asupra ratei de fetch. Abilitatea arhitecturii HSA de a forma run-time grupuri de instructiuni independente în bufferul de prefetch, face ca pierderea de performanta rezultata din faptul ca, rata de fetch este mai mica decât numarul pipe-urilor logice, sa nu fie resimtita în asa de mare masura asupra ratei de procesare.

La întâlnirea unei instructiuni de salt care se va executa (*taken branch*), valoarea registrului PC trebuie modificata astfel încât urmatoarea instructiune extrasa din cache sa fie facuta de la dresa destinatie a branch-ului. De asemenea, instructiunile aduse în buffer, ulterioare instructiunii de salt, trebuie evacuate deoarece acestea nu se vor mai executa. O instructiune de salt nu trebuie sa altereze valoarea registrului PC pâna când toate instructiunile, care vor umple "*branch delay slot*"-ul, nu vor fi aduse în bufferul de prefetch.

Când instructiunile sunt expediate cu succes din bufferul de prefetch spre unitatile functionale de executie, ele trebuie marcate drept "*evacuabile*" (squashed), grupuri contigue de astfel de instructiuni fiind apoi eliminate din buffer. Instructiunile valide ramase se vor deplasa înspre zona inferioara a bufferului, în cea superioara aducându-se alte FR instructiuni din cache. O caracteristica importanta a arhitecturii HSA consta în posibilitatea eliminarii codului executabil conditionat din bufferul de prefetch, într-un nivel pipeline anterior selectiei pentru expediere spre unitatile functionale de executie, existente în numar limitat. Totusi, nu trebuie eliminat codul conditionat care depinde de rezultatul unei instructiuni booleene neexecutate înca, din buffer. Prin aceasta evacuare se îmbunatateste mult rata de utilizare a setului de unitati functionale sporind si rata de procesare. Acest mecanism contrabalanseaza efectele negative ale expansiuni codului generat prin promovarea instructiunilor de pe calea eronata ("*untaken branch*"), pe care

nu va continua programul. Codul executabil conditionat poate fi marcat ca evacuabil fara a tine cont de pozitia lor în bufferul de prefetch. Instructiunile evacuabile, care se afla între instructiuni neevacuabile în buffer, nu sunt eliminate imediat deoarece contin si transmit informatia pozitionala, esentiala pentru functionarea corecta a mecanismului de întârziere a branch-ului. Valoarea numerica care însoteste branch-ul indica pozitia relativa a ultimei instructiuni dependente de cea de salt, astfel încât este esentiala pastrarea reprezentarii spatiale corecte în bufferul de prefetch (vezi tabelul 4.1).

Valoare numerică însoţitoare a instrucţiunii de salt						**Zonă de cod dependent de branch**				
	1	2	3	4	5	6	7	8	9	10
Condiţii						TB3	TB3	FB3		
Instrucţiuni	Instr1	Instr2	Instr3	Instr4	BRA (3)	Instr6	Instr7	Instr8	Instr9	Instr10
Evacuabile	E	E	E			E	E			
Instrucţiuni expediate spre unităţile funcţionale						B3 este evaluat ca "false"				

Tabelul 4.1.

Evacuarea codului din bufferul de prefetch

În tabelul 4.1, se prezinta comportarea instructiunilor în buffer. Primele trei instructiuni au fost expediate cu succes spre unitatile functionale de executie si sunt marcate drept "evacuabile". Instructiunile conditionate, pozitiile 6 si 7 în buffer, sunt de asemenea "evacuabile", întrucât registrul boolean B3 a fost evaluat deja ca fiind "false". Instructiunea de ramificatie, pozitia 5 în bufferul de prefetch, prin valoarea numerica însotitoare, specifica o zona dependenta de evaluarea branch-ului "*taken / non taken*", de trei instructiuni. Instructiunile evacuabile din pozitiile 6 si 7 nu sunt eliminate din buffer pâna când branch-ul nu este procesat de catre unitatea functionala corespunzatoare, întrucât prezenta lor e necesara pentru a identifica exact ultima instructiune dependenta de cea de salt.

Executia conditionata si speculativa

Implementarea executiei conditionate se face prin atasarea uneia sau mai multor variabile garda booleene, unei instructiuni date. Daca instructiunile executabile conditionat expediate spre unitatile functionale nu îndeplinesc toate conditiile impuse de variabilele garda booleene, unitatile functionale sunt reinitializate si devin disponibile pentru noi intrari în ciclul urmator de executie. Consideram urmatorul exemplu:

```
TB5  ADD R4, R5, #10     /* R4 = R5 + 10 */
TB5  LD R6, (R4, R7)     /* Încarca în R6 valoarea de la adresa data de
                            (R4+R7) */
FB6  ST (R4, R8), R9     /* Memoreaza R9 la adresa data de (R4+R8)*/
     ADD    R1, R1, #1   /* R1 = R1 + 1*/
```

Pentru executie, primele doua instructiuni necesita ca variabila booleana B5 sa fie evaluata ca “True”, iar cea de-a treia are nevoie ca registrul boolean B6 sa fie evaluat ca “False”. Ultima instructiune, negardata, se va executa indiferent de variabilele garda B5 si B6. Costul implementarii executiei gardate consta în biti suplimentari necesari codificarii informatiei garda în opcode-ul instructiunii. Unele metode codifica direct doar registrul boolean respectiv, în timp ce altele, pentru gardare folosesc o masca de biti a tuturor registrilor booleeni. Alegerea metodei folosite depinde de numarul maxim de variabile garda atasate unei instructiuni individuale si de dimensiunea setului de registri booleeni. Valorile booleene folosite pentru executia conditionata a instructiunilor sunt obtinute direct din file-ul de registri booleeni sau pot fi înaintate de catre unitatile functionale de executie la încheierea operatiilor logice sau relationale.

Instructiunile de salt conditionat deseori definesc doua cai de control a executiei programului, care se unesc ulterior dupa câteva instructiuni. Pentru separarea celor doua cai de urmat precum si pentru eliminarea instructiunilor de salt se introduce executia conditionata (gardata). Un exemplu sugestiv îl constituie constructia “*if-then-else*”.

```
if (R8 < 1)
    R1 = R2 + R3
else
    R1 = R5 – R7
    R10 = R1 + R11
```

În acest exemplu, valoarea continuta de registrul R8 determina care din cele doua instructiuni vor fi selectate pentru a calcula noua valoare a lui R1. Transpunerea în limbaj de asamblare HSA a exemplului anterior enuntat este urmatoarea:

```
        I1    LT B6, R8, #1
        I2    BF B6, dest1
        I3    ADD R1, R2, R3
        I4    BRA dest2
dest1:  I5    SUB R1, R5, R7
dest2:  I6    ADD R10, R1, R11
```

Instructiunea de salt conditionat I2, selecteaza pentru executiedintre instructiunile I3 si I5 în momentul executiei, în functie de valoarea registrului boolean B6. Executia conditionata permite eliminarea instructiunilor de salt I2 si I4, astfel:

	LT B6, R8, #1
TB6	ADD R1, R2, R3
FB6	SUB R1, R5, R7
	ADD R10, R1, R11

Transformarile efectuate asupra codului sunt benefice doar daca latenta totala a instructiunilor întâlnite pe calea determinata de branch pâna la punctul de reunificare a cailor de control a executiei, este mai mica decât latenta de penalizare a branch-urilor si daca prin eliminarea instructiunilor de salt se îmbunatateste timpul de executie.

Executia conditionata introduce variabilele garda, atasate instructiunilor, ce vor fi reorganizate de scheduller în zona de cod "*branch delay slot*". Unele tehnici de schedulling implica promovarea instructiunilor "de jos în sus", uneori putând depasi chiar instructiunea de salt conditionat, riscul fiind acela de a aplica tehnica de "*renaming*" asupra registrului destinatie. Codul care este promovat dincolo de instructiunea de salt conditionat se numeste **executat speculativ**, deoarece e posibil ca în momentul rularii programului (executiei branch-ului) saltul sa fie evaluat ca non-taken. Codul executat speculativ e marcat în acest sens de catre scheduller, facând disponibil procesorului aceasta informatie în momentul executiei. Marcarea unei instructiuni apartinând setului HAS, ca fiind speculativa, se face prin atasarea simbolului "**!**" opcode-ului instructiunii. Consideram urmatorul exemplu de cod executat speculativ:

Cod original	**Dupa promovare**	
SUB R1, R2, R3		SUB R1, R2, R3
LT B8, R1, #10		LT B8, R1, #10
BT B8, dest3	**FB8**	ADD**!** R7, R8, R1
ADD R7, R8, R1		BT B8, dest3
SUB R10, R7, R4		SUB R10, R7, R4

Codul promovat dincolo de instructiunea de salt poate fi însotit de aceeasi garda booleana ca cea folosita pentru controlul saltului, evitând astfel necesitatea de a redenumi registrul destinatie. Desi codul speculativ poate fi expediat spre unitatile functionale de executie, nu se permite

încheierea executiei acestuia pâna când branch-ul care determina calea de urmat, de unde este originar si codul speculativ, este taken.

Este important ca nici o eroare sau exceptie generata de executia unui cod speculativ (depasire aritmetica) sa nu întrerupa executia normala a procesorului pâna când nu este sigura necesitatea rezultatului executiei respectivului cod. De exemplu, este posibil ca executia speculativa a unei instructiuni Load sa cauzeze o eroare a magistralei de adresa. Daca aceasta instructiune cu referire la memorie este originara de pe o cale determinata de un branch, care în momentul executiei se evalueaza ca non-taken, în cazul respectivei erori nu se întreprinde nici o actiune. Registrul destinatie nu este marcat drept "invalid" si, doar daca o instructiune nespeculativa încearca sa acceseze acest registru ca operand sursa, se va genera o exceptie. Instructiunile speculative urmatoare celei cu referire la memorie, care acceseaza registrul alterat vor seta bitul de invalidare al registrilor destinatie aferenti noilor instructiuni, propagând potentialul de eroare pâna când problema va fi rezolvata.

Orice instructiune poate fi marcata drept speculativa, exceptând instructiunea Store, întrucât aceasta altereaza în permanenta starea masinii la fiecare scriere în memorie. O strategie posibila de executie speculativa a instructiunii Store consta în introducerea unui buffer de scriere (**Write Buffer**) pentru noile valori, mai degraba decât alterarea în mod direct a memoriei. Arhitectura HSA permite unei instructiuni Store sa promoveze înaintea unei instructiuni de salt conditionat, daca este gardata cu aceeasi variabila garda (conditie booleana) ca cea folosita pentru a selecta calea de urmat ("*branch path*") din care este originara instructiunea Store. Astfel de gardari (conditionari) pot limita promovarea ulterioara a instructiunii Store datorita dependentelor de date fata de instructiunea booleana care controleaza executia branch-ului.

Selectia si expedierea instructiunilor spre unitatile functionale de executie

Arhitectura HSA expediaza instructiunile din bufferul de prefetch spre unitatile functionale în ordinea fireasca a programului ("*in order*"). Prima instructiune neevacuata din partea inferioara a bufferului de prefetch este decodificata si primeste prioritatea cea mai mare, iar o alta instructiune neevacuata poate fi expediata concurent doar daca nu exista dependente de date cu membri grupului de instructiuni paralelizabile, existent deja. Numarul maxim de instructiuni care sunt expediate concurent este parametrizabil si determinat de numarul de pipe-uri logice asigurate în modelul arhitectural. În momentul determinarii unei dependente de date nici o alta instructiune nu se mai adauga grupului de instructiuni paralelizabile,

nepermitându-se expedierea instructiunilor “*out of order*”. Instructiunile din grupul creat sunt predate apoi spre executie unitatilor functionale corespunzatoare. Odata cu instructiunile (codul operatiei), unitatilor functionale le sunt transmise si operanzii sursa. Daca valoarea unui operand sursa nu este disponibila în file-ul de registri înseamna ca o noua valoare este calculata de catre o unitate functionala aferenta unei instructiuni anterior expediata. Daca unul din operanzii sursa este imposibil de obtinut, instructiunea aferenta este blocata si nici o alta instructiune nu mai este expediata în acest ciclu. Operanzii sursa specificati de catre o instructiune sunt identificati pe nivelul pipeline ID (decodificare) si pregatiti, pentru accesarea de catre unitatile functionale în timpul nivelului pipeline EX (executie). Este esential ca accesul la registri sa se faca pe nivelul ID, înainte de executia instructiunii, pentru a nu creste în mod nejustificat latimea perioadei de tact a procesorului.

Unitatile functionale gestioneaza un fond comun de resurse, care vor fi puse la dispozitia instructiunilor din grupul celor paralelizabile dupa principiul FIFO. Daca unei instructiuni nu i se poate asigura o unitate functionala de executie, respectiva instructiune este blocata si nici una din cele care i-ar urma nu mai sunt expediate spre executie. Orice instructiune care nu a fost expediata cu succes spre unitatile functionale vor ramâne în buffer si vor face parte din urmatorul grup paralelizabil de instructiuni decodificate, care vor fi trimise în ciclul urmator de executie.

Salturile întârziate reprezinta caracteristica esentiala a arhitecturii HSA, ce permite codului reorganizat generat de scheduller sa poata fi aplicat (transmis) unui sir de implementari diferite de procesoare, prin aceeasi secventa de cod. Valoarea definita de parametrul “*branch delay count*” [17] reprezinta numarul de instructiuni care trebuie executate înainte de executia codului de la adresa destinatie a saltului.

Procesarea instructiunilor de salt

Instructiunile de salt difera de celelalte tipuri de instructiuni prin faptul ca sunt executate în nivelul pipeline ID mai degraba decât în nivelul EX. La expedierea unei instructiuni Branch se detecteaza o unitate functionala corespunzatoare disponibila, iar procesarea se face în acelasi nivel ID. La evaluarea unei instructiuni drept non-taken, operatia este întrerupta, unitatea de branch este reinitializata, fiind disponibila pentru ciclul urmator de executie. La gasirea unui salt drept taken pot apare câteva situatii:

- Daca valoarea numerica ce însoteste instructiunea de salt este zero, înseamna ca nu exista instructiuni ce trebuie reorganizate, din zona dependenta de branch, iar bufferul de prefetch poate fi umplut cu

instructiuni începând cu locatia instructiunii de salt, ascendent. Registrul PC este actualizat cu adresa destinatie a instructiunii de salt. Instructiunile succesoare branch-ului, existente în bufferul de prefetch înainte de procesarea acestuia, sunt inhibate din procesarea lor. Unitatea functionala de branch îsi încheie activitatea.

- Daca valoarea ce însoteste instructiunea de salt este diferita de zero, ultima instructiune dependenta de cea de salt poate fi în buffer, pe magistrala în urma citirii din memorie, sau nu a fost extrasa din cache-ul de instructiuni.

În primul rând se determina daca ultima instructiune dependenta de salt se afla în buffer de prefetch. În acest caz se spune despre dependenta ca este "*satisfacuta*" si se pot aduce în siguranta instructiuni buffer începând cu prima locatie de dincolo de ultima instructiune dependenta, precum si se întrerupe procesul curent de fetch instructiune aflat în derulare. Instructiunile situate în acelasi grup paralelizabil cu instructiunea de salt, dar situate în buffer dincolo de zona dependenta de salt, vor fi inhibate în procesul lor de executie. Noua valoare a registrului Program Counter devine efectiva dupa ce unitatea de branch a încheiat procesarea.

Urmatorul caz ce va fi tratat este acela în care ultima instructiune dependenta de salt nu se gaseste în buffer, dar se afla pe magistrala ce uneste cache-ul de instructiuni cu procesorul (bufferul de prefetch). Instructiunile ce sosesc în buffer sunt înscrise începând cu prima locatie evacuabila pâna la locatia aferenta ultimei instructiuni dependente de salt. Executia instructiunilor din grupul celor decodificate (paralelizabile) continua nestingherita. Încheierea procesarii de catre unitatea de branch, face disponibila noua valoare a registrului PC.

Ultimul caz considerat este cel în care ultima instructiune dependenta de salt nu a fost extrasa înca din cache-ul de instructiuni. Se calculeaza numarul de pozitii ocupate din bufferul de prefetch care urmeaza instructiunii de salt si se adauga numarului de instructiuni care sunt citite din cache în acel moment. Numarul rezultat se scade din valoarea numerica ce însoteste instructiunea de salt pentru a afla numarul de instructiuni ("*short fall*") dependente de salt care trebuie extrase din cache. Aceasta valoare ("*short fall*") este retinuta de unitatea de branch, care continua procesarea saltului în perioadele de tact urmatoare. Pot trece mai multe perioade de tact pâna la detectarea ultimei instructiuni dependente de salt, si în timpul fiecarei perioade valoarea "*short fall*" este recalculata. Doar când "*short fall*" devine zero, noua valoare a registrului PC poate avea efect si unitatea de branch încheie procesarea.

Este posibila reorganizarea codului de catre schedullerul HSS, aferent arhitecturii HSA, astfel încât instructiuni de salt sa se regaseasca în zone

dependente de un salt anterior. În momentul în care grupul de instructiuni decodificate, paralelizabile, sunt expediate spre unitatile functionale de executie, pot exista mai multe instructiuni de salt în acelasi grup. Atâta timp cât exista suficiente unitati de branch disponibile, toate instructiunile de salt pot fi expediate spre executie, functie doar de dependentele de date existente. Orice numar de salturi evaluate ca non-taken pot fi procesate în paralel, pâna la inclusiv primul branch gasit taken. Odata detectat un branch taken, valoarea ce însoteste instructiunea de salt poate inhiba executia instructiunilor din acelasi grup paralelizabil. Daca un al doilea branch, din acelasi grup paralelizabil de instructiuni decodificate, este evaluat ca fiind taken, nu este permis ca branch-ul sa aiba efect în ciclul curent, chiar daca valoarea numerica ce însoteste instructiunea de salt este cunoscuta. În fiecare ciclu, doar o instructiune de salt evaluata ca fiind taken poate altera registrul PC. A doua instructiune de salt, taken fiind, este tratata în ciclurile ulterioare, când valoarea numerica ce însoteste branch-ul este reevaluata pentru a determina daca zona de cod dependenta de salt este cunoscuta. Aceasta decizie depinde de modificarea starii bufferului de prefetch si de gradul de ocupare al unitatii de fetch, cauzate de procesarea primului salt si a instructiunilor dependente de primul salt. Primul salt poate determina ca, instructiunile din buffer dependente de al doilea salt, sa fie eliminate. Astfel, este indicat ca, o instructiune de salt, aflata în zona de cod dependenta de un salt anterior, sa fie ea însasi urmata de o zona de cod dependenta, de dimensiune cel putin egala cu zona de cod dependenta, aferenta primului salt.

Unitatile functionale si partajarea resurselor

Setul de instructiuni al arhitecturii HSA este divizat în doua tipuri, în functie de natura destinatiei fiecarei instructiuni. Aceasta clasificare simplifica implementarea hardware a procesorului prin gruparea fizica a unitatilor functionale specifice unui tip de instructiuni în jurul file-urilor de registri cu care ele interactioneaza. Prin aceasta abordare se reduce complexitatea interconectarii necesara într-un procesor puternic paralel. Arhitectura HSA este caracterizata de urmatoarele tipuri de unitati functionale de executie:

- **Aritmetica**
- **Multiplicativa**
- **Load** (citire din memorie)
- **Branch** (de salt)
- **Relationala** (booleana)
- **Shift** (de deplasare si rotire)
- **Store** (scriere în memorie)

Numarul maxim de instructiuni care pot fi expediate spre executie într-un ciclu variaza între implementarile diferite de procesoare, în functie de numarul de pipe-uri logice disponibile. Pipe-urile servesc la rutarea instructiunilor, împreuna cu operanzii lor sursa, spre unitatile functionale corespunzatoare. Prin asigurarea unui fond comun de unitati functionale disponibile tuturor pipe-urilor, se realizeaza o economie de resurse. Se realizeaza un compromis între reducerea complexitatii de implementare *on chip* a arhitecturii si cresterea riscului de blocare a procesarii (hazarduri structurale) datorita lipsei de resurse. Este de dorit ca numarul de unitati functionale sa fie pastrat cât mai mic posibil pentru reducerea problemelor hardware de interconectare, si pentru utilizarea eficienta a resurselor cât mai mult timp posibil (procesarea de cod folositor de catre unitatile functionale).

Toate unitatile functionale sunt disponibile în forma pipeline sau non-pipeline. Unitatile non-pipeline pot procesa doar o instructiune la un moment dat si sunt indisponibile pentru o nou intrare pâna când instructiunea curenta s-a încheiat de executat. Unitatile pipeline sunt disponibile întotdeauna pentru o noua intrare în fiecare perioad de tact, doar daca nu se afla "în asteptare" pentru ocuparea magistralei rezultat. O exceptie de la cele enuntate anterior o reprezinta unitatea aritmetica de procesare a instructiunilor de împartire cu operanzi întregi. Întrucât frecventa de aparitie a instructiunilor de împartire în programele de uz general este redusa [8], nu se justifica necesitatea unei unitati functionale speciale, dedicata împartirii. În schimb, unitatea aritmetica este fortata sa adopte mai degraba latenta asociata instructiunii de împartire decât latenta aritmetica uzuala. O unitate aritmetica pipeline va functiona ca o unitate non-pipeline la procesarea instructiunii de împartire, semnificând faptul ca, intrarile unitatii functionale sunt blocate pentru mai multi cicli, în functie de diferenta relativa a latentei instructiunilor.

Unitatile de branch difera de celelalte unitati functionale prin faptul ca ele pot încheia procesarea instructiunii în nivelul pipeline ID si au latenta fixa, unitara. Aceasta reprezinta valoarea minima a latentei unitatii functionale branch, întrucât încheierea procesarii poate dura mai multe perioade de tact daca trebuie sa astepte pentru aducerea tuturor instructiunilor dependente de salt în bufferul de prefetch.

Ultima sarcina a unitatilor functionale de executie o reprezinta scrierea rezultatului, pe nivelul pipeline **WB** (write back), în file-ul de registri corespunzator. Daca o instructiune ulterioara necesita respectivul rezultat ca si operand sursa, noua valoare este înaintata unitatii functionale care emite cererea, fara a mai astepta încheierea procesului de scriere în registri. Totusi, numarul magistralelor rezultat pentru valori întregi este limitat în modele particulare de procesare, fiind necesara o arbitrare a unitatilor functionale

care vor sa acceseze respectiva magistrala. Doar unitatile carora le-au fost alocate magistrala rezultat pot sa-si încheie operatiile interne si sa paseze rezultatul altor unitati fuctionale. Unitatile care nu au primit acceptul de accesare a magistralei rezultat sunt stagnate în executia lor, dar beneficiaza de tratament preferential la urmatoarea runda de arbitrare. Unitatile functionale pipeline care au iesirea (rezultatul) în stare de asteptare vor fi disponibile sa preia spre procesare noua instructiune în ciclul urmator de executie doar daca nivelele pipeline nu sunt toate deja ocupate. Arhitectura HSA ofera un numar suficient de magistrale rezultat booleene, nefiind necesara o arbitrare la nivel de magistrala pentru unitatile care opereaza asupra instructiunilor booleene (logice).

Setul de instructiuni al procesorului superscalar HSA

Setul de instructiuni al arhitecturii HSA se bazeaza pe setul de instructiuni al procesorului HARP [13] si cuprinde toate formatele de instructiuni HARP posibile. Pe lânga acestea, distingem instructiuni complexe, care implementeaza operatii combinate implicând trei operanzi sursa, sau operatii în virgula mobila. Toate instructiunile pot fi multiplu gardate si marcate pentru executie speculativa, exceptând scrierile în memorie. Instructiunile oferite de arhitectura HSA sunt caracterizate de latente diferite, variabile ca marime. Majoritatea instructiunilor au latenta unitara (o perioada de tact). Instructiunea aritmetica de împartire (DIVIDE) are latenta tipica de 16 perioade de tact. Înmultirile sunt efectuate în unitati aritmetice dedicate (multiplicative) si au latenta de 3 perioade de tact. Instructiunile cu referire la memorie si cele de salt au latenta dependenta de timpul de acces la ceche-urile de date respectiv instructiuni, primul tip de instructiuni fiind influentat si de modul de adresare adoptat (adresare OR).

Pe lânga instructiunile **aritmetico-logice** uzuale, specifice procesoarelor RISC (vezi [15], Cap. 5 – *Arhitectura Microprocesoarelor Mips R2000/R3000*), de adunare, scadere, înmultire, împartire, SI/SAU logic, deplasare aritmetico-logica, cu operanzi registri, si/sau valoare imediata, cu sau fara semn, distingem si instructiuni de extensie semn, instructiuni logice combinate.

Instructiunile **relationale**, cu sau fara semn, seteaza / reseteaza registri booleeni, acestia fiind folositi ulterior în cadrul instructiunilor de salt. Totodata, registrii booleeni pot fi alterati de catre instructiunile booleene, de cele de transfer sau chiar de cele cu referire la memorie. În ultimele doua cazuri, registrii booleeni sunt înscrisi cu cel mai putin semnificativ bit al registrului sursa, sau al locatiei de memorie citite. Exista, de asemenea, instructiuni speciale de **validare** / **invalidare întreruperi**, **exceptii software**.

Instructiunile **cu referire la memorie** (load / store) se caracterizeaza prin faptul ca pot avea unul sau doi operanzi sursa. Registrul destinatie poate fi fie un registru întreg fie unul boolean. Atunci când operandul sursa este compus din doi registri efectul este înscrierea a pâna la patru registri generali (citire pe cuvânt = 4octeti, pe dublu cuvânt si pe patru cuvinte) sau scrierea în memorie a pâna la 16 octeti (scriere un cuvânt, dublu cuvânt sau patru cuvinte) de date.

Instructiunile **în virgula mobila** (aritmetico-logice, de transfer, relationale si cu referire la memorie) sunt similare cu cele aferente numerelor întregi, diferenta fiind ca operanzii sunt registrii flotanti (simpla sau dubla precizie).

4.3. OPTIMIZAREA STATICA A PROGRAMELOR

4.3.1. INTRODUCERE

Scheduler-ul dezvoltat la Universitatea din Hertfordshire, UK (HSS - Hatfield Superscalar Scheduler) [28] a fost implementat ca parte integranta a proiectului HSA (Hatfield Superscalar Architecture) – o arhitectura superscalara minima care combina cele mai bune caracteristici ale conceptelor VLIW si superscalare [30]. Obiectivul HSS este atingerea unor performante cu un ordin de marime mai mari comparativ cu un procesor scalar RISC clasic, evitând totodata cresterea exploziva a codului.

HSS rearanjaza codul HSA scris în limbaj de asamblare [14, 7] pentru a forma grupuri de instructiuni care pot fi expediate în paralel unitatilor functionale în momentul executiei. Scheduling-ul instructiunilor poate fi vazut ca un proces în care fiecare instructiune se încearca a fi succesiv mutata sau "*infiltrata în sus*" [23] prin structura de cod într-o încercare de a fi executata cel mai devreme posibil. Acest proces este stopat de catre dependentele de date dintre instructiuni.

4.3.2. HSS ÎN CONTEXTUL ACTUAL

Trace Scheduling [21], care este poate cea mai cunoscuta tehnica de scheduling, a fost dezvoltata de catre J. Fisher pe baza unei arhitecturi VLIW (Bulldog). La baza acestei tehnici sta conceptul de *trace* – care reprezinta o cale printr-o secventa de basic block-uri. Trace-urile sunt selectate si reorganizate în ordinea frecventei lor de executie. Trace-ul selectat este reorganizat ca si cum ar fi un singur basic block. La intrarea si iesirea din trace este adaugat codul necesar pentru a pastra semantica programului dupa scheduling. Procesul se repeta pâna când tot codul este optimizat în vederea executiei. Totusi, codul din trace-uri succesive nu se suprapune niciodata si nu se aplica tehnica *software pipelining* [19, 22, 25]. Aceasta problema poate fi rezolvata prin desfasurarea buclelor ("*loop unrolling*"), obtinându-se un trace mai mare, în care corpul buclei pentru iteratii diferite este executat în paralel. Dezavantajul este ca loop unrolling este un mecanism care implica explozia codului.

Tehnica introdusa de Fisher poate fi aplicata la fel de bine si procesoarelor superscalare cu executie *in order*. Sunt necesare doar usoare modificari pentru a pastra semantica la nivelul instructiunii si a se asigura executia codului rezultant si în stilul secvential original.

Dezvoltarile actuale în scheduling încearca sa aplice tehnica software pipelining si se bazeaza fie pe *Tehnica Modulara de Scheduling* [26, 25] a lui Rau si Fisher fie pe algoritmul *Enhanced Percolation Scheduling* [20] dezvoltat de catre K. Ebcioglu la IBM. Software pipelining este o metoda de suprapunere a instructiunilor din iteratii diferite ale buclelor, fara derularea initiala a buclei, în scopul minimizarii numarului de cicli între iteratiile succesive ale buclei.

Schedulingul modular se bazeaza pe un **i**nterval de **i**nitiere (II) fixat. II reprezinta întârzierea dintre începutul iteratiilor succesive ale buclelor. Scheduling-ul modular calculeaza o limita inferioara pentru II. Valoarea minima pentru II poate fi determinata din bucla dependentelor de date. Bucla dependentelor de date (figura 4.2) contine dependentele dintre instructiuni aflate în iteratii diferite ale buclei. Lantul dependentelor de date din figura 4.2 implica un II de minim 4 instructiuni. II poate fi, de asemenea, limitat de numarul de instructiuni care pot fi lansate în executie în fiecare ciclu. De exemplu, daca exista 5 instructiuni aritmetice în bucla si doar doua operatii ALU pot fi lansate în executie în fiecare ciclu, II trebuie sa fie minim 3. Daca II ar fi 2, în conditiile de mai sus, nu ar aparea hazard (nici macar structural).

Se va crea astfel, o bucla de optimizat – numita *fereastra* – având un numar de instructiuni egal cu valoarea minima a lui II. O tabela de rezerva este folosita pentru a înregistra resursele folosite în mod curent de catre instructiunile din bucla respectiva. Unele instructiuni pot fi lasate în pozitia initiala din program ("neschedulate") datorita conflictelor la resurse si reordonate ulterior în fereastra, ca parte a algoritmului de backtracking aplicat. Când o instructiune este marcata ca “neschedulata”, informatiile despre resursele apartinând acelei instructiuni sunt înlaturate din tabela de rezerva. Daca fereastra nu poate fi schedulata pentru un II dat, II este incrementat si procesul se reia pâna când se obtine o bucla schedulata satisfacator. Dupa scheduling pot aparea zone de cod anterioare si / sau posteriore buclei schedulate.

Schedulingul modular functioneaza doar pe basic block-uri separat. Principala provocare este de a extinde aplicarea algoritmului la bucle cu structura arbitrara oricât de complexa.

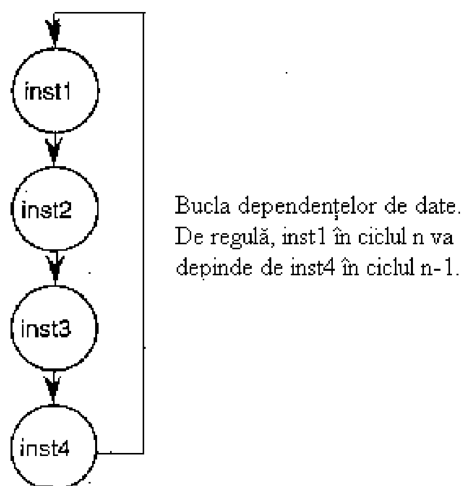


Figura 4.2. Bucla dependentelor de date

O alta tehnica propusa este executia gardata (conditionata prin variabile booleene de garda) a basic block-urilor multiple într-un singur basic block înainte de scheduling. Tehnica se numeste *if – conversie* [32], însa codul schedulat rezultat este departe de a fi optim.

În contrast cu schedulingul modular, algoritmul *Enhanced Percolation Scheduling* pastreaza corpul buclei intact pe parcursul procesului de scheduling.

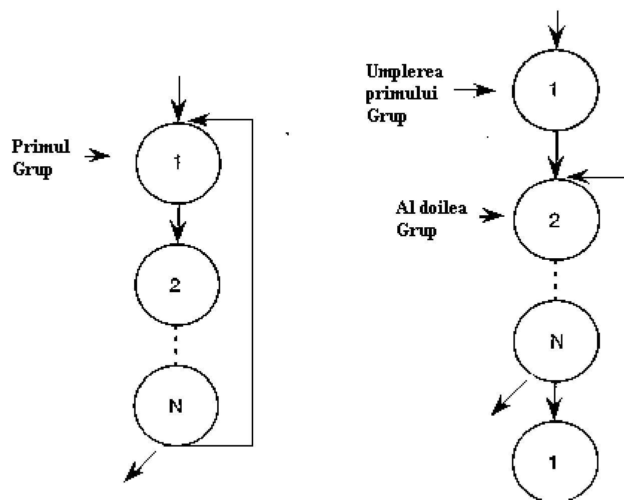


Figura 4.3. Algoritmul Enhanced Percolation Scheduling

Un grup de instructiuni, numit "*fence*" este selectat si sunt cautate instructiuni care pot coexista cu grupul selectat. Procesul continua până când se umple grupul sau până când nu mai gasim astfel de instructiuni. Instructiuni succesoare grupului deja creat vor forma noi grupuri. Instructiunile din grupurile deja create pot migra spre începutul buclei. Pentru a facilita aceasta miscare de cod sunt introduse doua copii ale primului grup, una la intrarea si alta la iesirea din bucla (figura 4.3). Copia de la sfârsitul buclei reprezinta operatiile din iteratia urmatoare a buclei. Întregul proces este repetat până când toate instructiunile au fost mutate în grupuri. O provocare referitor la aceasta tehnica o constituie evitarea expansiunii codului cauzata de duplicarea agresiva a unor astfel de grupuri.

Bazat pe acest algoritm HSS poate reorganiza bucle de orice complexitate. Exista totusi o diferenta privind doua aspecte: primul, fiecare instructiune este reordonata si se încearca infiltrarea ei în sus prin structura de cod cât mai mult posibil; alegerea instructiunilor candidate se face doar pentru a vedea daca pot fi incluse în grupul de instructiuni curent asamblate. Al doilea aspect se refera la restrictionarea traversarii codului dincolo de punctul de start al buclei. HSS permite utilizarea tehnicii software pipelining [22].

4.3.3. MECANISMUL DE REORGANIZARE SI OPTIMIZARE

HSS a fost dezvoltat pentru îmbunatatirea performantei prin schedulingul static al instructiunilor. Înaintea procesului de scheduling, HSS introduce informatiile aferente programelor de test în structurile de date corespunzatoare. Schedulerul citeste cuvinte de instructiuni – LIW (long instruction word), detecteaza basic block-uri, tintele branch-urilor, proceduri, bucle, bucle imbricate si calculeaza durata de viata a registrelor (numarul de instructiuni aflat între instructiunea care înscrie un registru si ultima instructiune care îl utilizeaza pe post de sursa). Exista suplimentar posibilitatea de a aplica *inlining functiilor*.

La un moment dat, programul reorganizeaza o singura procedura. Deoarece programele pierd mult timp în bucle care asigura un mare potential de paralelism, buclele imbricate sunt optimizate primele, urmate de buclele exterioare si apoi de restul de cod.

HSS este un scheduler parametrizabil caracterizat de doua tipuri de parametri de configurare. Primul tip selecteaza modelul de arhitectura (de exemplu, numarul si tipul unitatilor de executie). Al doilea tip de parametri specifica diversele optiuni de scheduling disponibile (de exemplu, decizia ca un bloc de instructiuni sa poata - sau nu - fi promovat în sus, într-un basic block încheiat cu o instructiune BSR).

Mecanismul de scheduling HSS examineaza tehnica de infiltrare locala si globala a instructiunilor. Aceasta implica doua procese majore: verificarea apartenetei instructiunilor la grupurile de instructiuni deja create, si în caz afirmativ, determinarea posibilitatii ca respectivele instructiuni sa treaca în grupul urmator de instructiuni.

Structura HSS (figura 4.4) are la baza algoritmul “Backedge” si înglobeaza ambele tehnici de infiltrare (percolare) locala si globala. Întâi este apelata cea locala, apoi cea globala pentru a determina daca o instructiune poate trece într-un alt basic blok situat anterior. Se verifica totodata posibilitatea combinarii instructiunilor (*merging*) – tehnica folosita pentru eliminarea dependentelor reale de date (RAW) precum si analiza aliasurilor la memorie în cazul existentei acestora (adresele instructiunilor cu referire la memorie – Load/Store).

Verificarea coexistentei instructiunilor

Se face în timpul procesului de infiltrare. O instructiune, care urmeaza a fi infiltrata în sus în structura de cod obiect poate coexista în interiorul unui grup de instructiuni daca nu exista dependente reale de date între respectiva instructiune si cele deja existente în grup si daca exista suficiente

unitati functionale disponibile. Considerând doua instructiuni, *inst1* deja existenta în grupul de instructiuni, si *inst2* – cea care va fi infiltrata, daca apare o antidependenta (hazard WAW) între cele doua instructiuni nu este nici o problema deoarece inst2 este adaugata implicit la sfârsitul grupului de instructiuni. Acolo unde este posibil dependentele sunt tratate prin stergerea lui inst1. Însa daca inst2 este gardata ea nu se executa neaparat întotdeauna daca inst1 se executa, caz în care nu este bine sa se stearga inst1. În cazul prezentei hazardurilor RAW acestea pot fi eliminate fie prin tehnica "*merging*" fie prin colapsare dinamica a instructiunilor [27, 33].

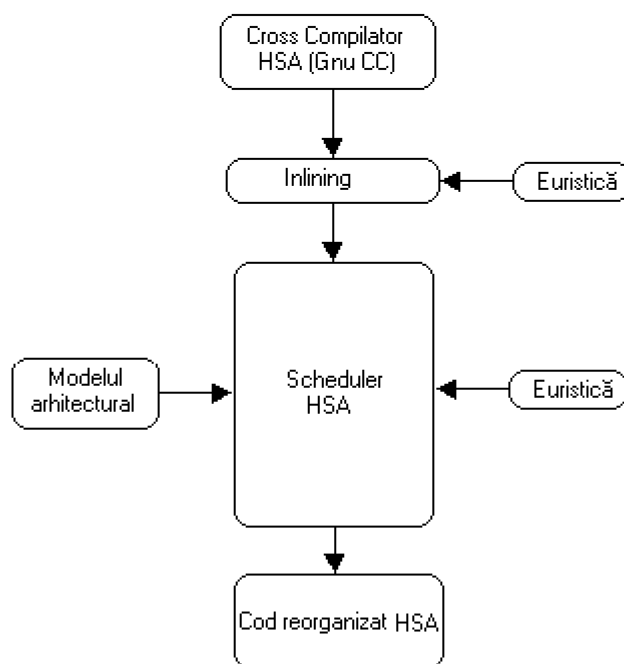


Figura 4.4. Structura schedulerului HSS

Verificarea posibilitatii trecerii instructiunilor

Daca apare o antidependenta între un grup de instructiuni si o instructiune care se infiltreaza atunci registrul destinatie al instructiunii respective trebuie renumit înainte de a trece de instructiunea din grup care a cauzat ambiguitatea. Este apelata tehnica de analiza anti-alias la memorie care compara cele doua adrese si în caz ca acestea difera este permisa trecerea instructiunii care se infiltreaza dincolo de instructiunea din grup.

Infiltrarea individuala a instructiunilor

La infiltrarea individuala a unei instructiuni trebuie examinata posibilitatea de deplasare atât prin basic block-ul curent cât si prin blocurile succesoare celui curent. De regula, o instructiune trebuie sa fie capabila sa se infiltreze pe mai multe cai. Schedulerul trebuie sa lucreze cu copii multiple ale instructiunii percolante care pot patrunde în acelasi basic block pe cai diferite. În consecinta, trebuie rezolvate aceste probleme pentru pastrarea corecta a semanticii programului.

Infiltrarea basic block-urilor

Consideram urmatorul exemplu:

LIW0: SUB R_2, R_3, #4
LIW1: LD R_1, (R_0, R_5)
LIW2: MOV R_7, R_2; ADD R_2, R_3, R_4
inst3: ADD R_5, R_1, R_6 /* instructiunea care se infiltreaza */

Exemplul respectiv arata ca desi instructiunea inst3 poate trece de toate instructiunile din LIW2 datorita dependentei RAW dintre inst3 si instructiunea LD din LIW1, inst3 nu poate promova mai sus. Inst3 poate coexista în interiorul grupului LIW2 astfel:

LIW0: SUB R_2, R_3, #4
LIW1: LD R_1, (R_0, R_5)
LIW2: MOV R_7, R_2; ADD R_2, R_3, R_4, ADD R_5, R_1, R_6
inst3:

Daca o instructiune ajunge sa fie prima într-un basic block atunci ea se infiltreaza atât în basic block-ul predecesor cât si în basic block-ul destinatie. Instructiunile care se infiltreaza trebuie sa reuseasca în toate blocurile atât cel predecesor cât si cel destinatie pentru a pastra validitatea semanticii programului. Tehnica "*percolation*" poate implica, inserarea uneia sau a mai multor versiuni ale instructiunii originale si stergerea celei originale din scheduler. Exista doua aspecte majore ce trebuie rezolvate de catre scheduler. Primul, daca o instructiune merge pe mai multe cai distincte, si întâlneste ulterior tot o versiune a ei (fie aceasta copia nr. 2) într-un basic block, ea poate fi cu succes inserata în acel bloc daca orice garda booleana sau orice alteratii aparute în timpul infiltrarii permit ambelor copii ale instructiunii sa fie combinate pentru a forma o singura instructiune; în caz contrar infiltrarea primei copii a instructiunii va esua. Al doilea aspect

se refera la faptul ca, daca o copie a unei instructiuni s-a infiltrat deja cu succes în tot basic block-ul de pe o anumita cale, atunci cea de-a doua copie a instructiunii urmând o alta cale nu poate fi inserata în acel basic block. De asemenea, aceasta trebuie sa strabata prin tot block-ul, altfel procesul esueaza.

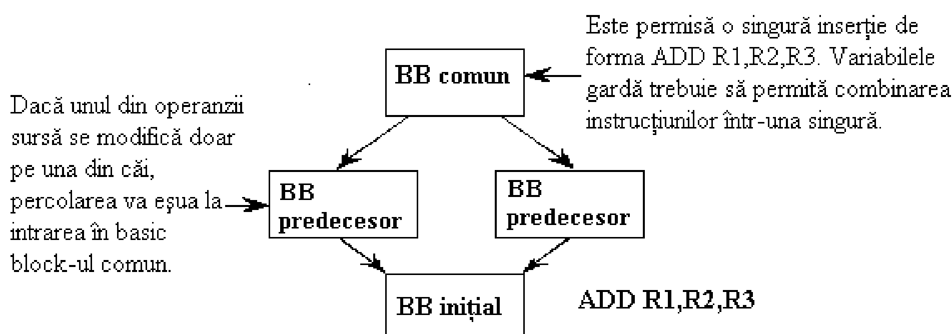


Figura 4.5. Combinarea instructiunilor într-un Basic Bloc Comun

Executia conditionata a instructiunilor

La intrarea într-un nou basic block, pot fi adaugate variabile de garda booleene daca fluxul de control al programului din block este determinat de un salt conditional (**BT** pe "true"sau **BF** pe "false"). Presupunem ca avem instructiunea: **BT B1, Label**. Daca blocul predecesor este urmatorul în secventa atunci acestuia i se adauga garda opusa **F B1**. Pe de alta parte, daca blocul predecesor este un bloc destinatie atunci se adauga garda **T B1**. Numarul variabilelor garda conjugate (*AND*) este setat de catre utilizator. Astfel, la promovarea unei instructiuni printr-o serie de basic block-uri implica adaugarea de garzi multiple. Consideram, drept exemplu, urmatoarea secventa:

```
      NE B1, R11, R12
      BT B1, L8
      …
      ADD R10, R6, # -1 ; secventa succesoare care se deplaseaza
L8:   MOV R5, #6        ; secventa destinatie care se deplaseaza
```

Dupa reorganizare secventa devine:

```
      NE B1, R11, R12
      BT B1, L8
FB1   ADD R10, R6, # -1
TB1   MOV R5, #6
```

În timpul infiltrarii o instructiune poate capta un numar mai mare decât cel permis de garzi. Daca instructiunea nu este un *store* sau un *branch* atunci variabila de garda care a fost captata cel mai devreme este înlocuita si registrul destinatie este redenumit. Pentru instructiunile de salt si scriere în memorie problema este tratata diferit. În primul rând o instructiune de salt nu trebuie sa piarda vreo garda. În al doilea rând, o instructiune store nu poate fi executata în mod speculativ, deoarece s-ar altera iremediabil contextul programului.

Tratarea instructiunilor cu latente mari

Instructiunile a caror executie necesita mai mult de un ciclu se numesc instructiuni "cu latente mari". Astfel de instructiuni sunt cele de înmultire, împartire, flotante si cele cu referire la memorie. Numarul de cicli în care se executa o instructiune *load* depinde de numarul de cicli necesari accesarii datei din cache-ul de date sau din memoria interna. Când o astfel de instructiune se infiltreaza într-un nou grup de instructiuni, acest grup trebuie sa nu contina instructiuni care folosesc ca sursa registrul destinatie al instructiunii load. În timpul rularii procesul va stagna pâna când executia instructiunii *load* se va fi încheiat, scheduler-ul nefiind unul optim.

Exemplu:

LIW1: ADD R1, R2, R3; LD R5, (R0,R6)
LIW2: SUB R8, R5, R4

Pentru obtinerea unui scheduler optim trebuie inserat un grup de instructiuni suplimentar. Optimizarea HSS se face prin inserarea unei copii virtuale (**VCOPY**) între cele doua grupuri de instructiuni, necesara terminarii executiei instructiunii cu latenta mare. Copiile virtuale au forma **VCOPY Ri, Ri –** unde Ri este registrul destinatie al instructiunii cu latenta mare. Acestea nu folosesc nici o resursa si nici nu sunt incluse în codul final schedulat. Scopul lor este de a pune în valoare instructiunile cu latenta mare în timpul procesului de scheduling. Exemplul de mai sus devine:

LIW1: ADD R1, R2, R3; LD R5, (R0,R6)
LIW2: VCOPY R5, R5
LIW3: SUB R8, R5, R4

VCOPY asigura ca instructiunea SUB nu poate fi mutata în grupul LIW2 datorita dependentelor reale de date prin registrul R5.

Instructiunile VCOPY sunt tratate exact la fel ca celelalte instructiuni si pot promova la rândul lor în sus în structura de cod. Probleme apar când instructiunile cu latenta mare se infiltreaza pe cai diferite în câteva basic block-uri distincte. Daca VCOPY este generata o singura data atunci ea nu poate fi utilizata de catre toate noile instructiuni cu latenta mare inserate. Ulterior, ele tind sa se separe de instructiunile cu latenta mare corespunzatoare în timpul scheduling-ului. Separarea se face prin *renaming* aplicat atât instructiunii cu latenta mare cât si copiei virtuale VCOPY.

Versiunea noua a HSS genereaza o instructiune VCOPY pentru fiecare instanta a instructiunii cu latenta mare. Acest lucru implica refacerea cailor de infiltrare prin câteva basic block-uri. Totodata stergerea unei instructiuni cu latenta mare din scheduler determina stergerea copiei asociate VCOPY.

Urmatorul exemplu arata cum sunt inserate instructiunile VCOPY când o instructiune de înmultire se infiltreaza cu succes în doua basic block-uri, ca în figura de mai jos.

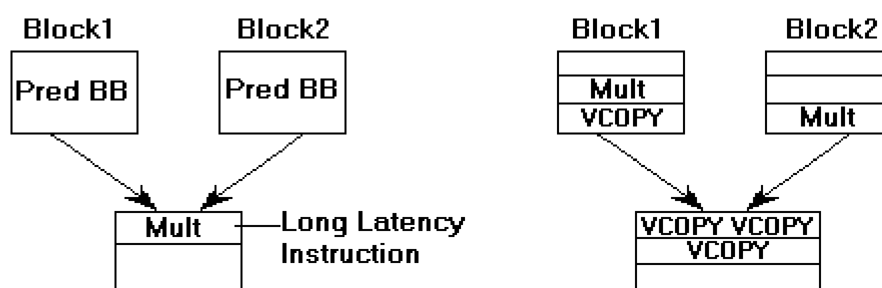


Figura 4.6. Inserarea unei instructiuni VCOPY dupa o instructiune cu latenta mare

O instructiune de înmultire se executa pe procesorul HSA în trei cicli si de aceea necesita doua instructiuni VCOPY. În basic block-ul 1 instructiunea de înmultire s-a infiltrat în penultimul grup de instructiuni iar în basic block-ul 2 în ultimul grup. Copiile virtuale sunt introduse astfel: blocul 1 are o instructiune VCOPY în ultimul grup iar cealalta copie se afla în primul grup al blocului succesor care este de fapt blocul de unde a promovat instructiunea de înmultire originala. În blocul 2 deoarece instructiunea de înmultire se afla în ultimul grup cele doua copii vor fi inserate în blocul succesor în primul si al doilea grup de instructiuni. Daca instructiunea de înmultire se reinfiltreaza cu succes în alte blocuri, noi instructiuni VCOPY vor fi inserate odata cu ea, iar copiile originale aferente instructiunii de înmultire anterior introduse pe aceeasi cale vor fi sterse.

Infiltrarea instructiunilor de salt

HSS considera drept "*Delay Slot*" numarul de grupuri de instructiuni situate dupa grupul care contine instructiunea de salt si care trebuie executate înainte ca branch-ul sa fie executat. HSS trateaza instructiunile de salt care promoveaza în sus similar cu celelalte instructiuni: verificarea dependentele si antidependentele de date, verificarea apartenetei instructiunilor la grupurile de instructiuni deja create, si în caz afirmativ, determinarea posibilitatii ca respectivele instructiuni sa treaca în grupul de instructiuni urmator. Totusi, infiltrarea instructiunilor de salt difera de celelalte în doua aspecte importante. Primul, este acela ca salturile pot urca în program doar cu un numar de grupuri de instructiuni maxim egal cu parametrul Delay Slot. Al doilea aspect se refera la faptul ca, daca un salt (fie acesta al doilea branch) urca într-un basic bloc predecesor care contine deja o instructiune de salt (primul branch), atunci al doilea branch devine instructiune în primul *branch delay slot* si celelalte instructiuni din primul delay slot care urmeaza celui de-al doilea branch vor fi incluse în cel de-al doilea *branch delay slot*.

Fuzionarea instructiunilor (*merging*)

HSS foloseste tehnica "*merging*" pentru a depasi limitarile introduse prin dependentele reale de date. Aceasta implica combinarea a doua instructiuni într-una singura. Exista trei categorii de astfel de instructiuni. Prima categorie, numita **MOV Merging** implica o pereche de instructiuni în care prima din ele este MOV. A doua categorie numita **Immediate Merging** se caracterizeaza prin faptul ca ambele instrutiuni au ca operanzi sursa valori immediate. A treia categorie se numeste **MOV Reabsorption** si are ca a doua instructiune o instructiune MOV sau instructiunea ce se va infiltra va fi convertita la tipul primei instructiuni.

❖ MOV Merging

Când apare o dependenta reala de date între o instructiune MOV si o instructiune care încearca sa promoveze în sus în structura de cod a programului, se verifica faptul daca cele doua instructiuni pot fi procesate în paralel. În caz afirmativ instructiunea ce se infiltreaza îsi continua drumul ascendent prin basic block. În continuare vom prezenta tipurile de situatii ce pot sa apara în cadrul tehnicii MOV merging.

- Combinarea cu o instructiune de **adunare**.

a) Secventa initiala:

```
MOV R6, R7
ADD R3, R6, R5    /* instructiunea care se infiltreaza */
```

Secventa modificata:

MOV R6, R7; ADD R3, R7, R5

b) Secventa initiala:

MOV R6, #4
ADD R3, R6, #5 /* instructiunea care se infiltreaza */

Secventa modificata:

MOV R6, #4; MOV R3, #9

Prin înlocuirea registrului R6 cu valoarea imediata respectiva instructiunea de adunare devine MOV.

- Combinarea cu o instructiune **store**

Secventa initiala:

MOV R3, #0
ST (R1, R2), R3 /* instructiunea care se infiltreaza */

Secventa modificata:

MOV R3, #0; ST (R1, R2), R0

Registrul R3 este înlocuit cu R0 deoarece toate procesoarele RISC au registrul R0 cablat la 0.

- Combinarea cu o instructiune **relationala**

Secventa initiala:

MOV R4, #4
GT B1, R4, R3 /* instructiunea care se infiltreaza */

Secventa modificata:

MOV R4, #4; LTE B1 R3, #4

Registrul R4 este înlocuit cu valoarea imediata memorata de el si instructiunea GT devine LTE pentru a permite operanzilor instructiunilor interschimbarea.

- Combinarea instructiunilor **gardate**

a) Secventa initiala:

EQ B3, R0, R0 /* B3 := true */
TB3 ADD R10, R11, R12/* instructiunea care se infiltreaza */

Secventa modificata:

EQ B3, R0, R0; ADD R10, R11, R12

Instructiuni de tipul EQ Bi, R0, R0 si NE Bi, R0, R0 sunt folosite pentru a înlocui instructiunile MOV Bi, #true sau MOV Bi, #false pe care arhitectura HSA nu le pune la dispozitie. Deoarece B3 este întotdeauna *true* garda TB3 aferenta instructiunii de adunare poate fi înlaturata. Daca B3 ar fi

evaluata întotdeauna la *false* atunci instructiunea care se infiltreaza va fi înlocuita cu un NOP.

b) Secventa initiala:

MOV B1, B2
TB1 LD R4, (R0, R6) /* instructiunea care se infiltreaza */

Secventa combinata:

MOV B1, B2; **TB2** LD R4, (R0, R6)

Pentru eliminarea dependentelor de date, garda instructiunii LD devine acum B2.

c) Secventa initiala:

MOV B1, B2
BT B1, label /* instructiunea care se infiltreaza */

Secventa modificata:

MOV B1, B2; BT B2, label

d) Secventa initiala:

EQ B1, R0, R0
BT B1, label /* instructiunea care se infiltreaza */

Secventa modificata:

BRA label

Daca registrul boolean care constituie garda pentru instructiunea care se infiltreaza are valoare constanta saltul fie va fi eliminat fie va fi transformat într-unul neconditionat (BRA).

❖ **Immediate Merging**

Aceasta tehnica implica orice pereche de instructiuni care au valori imediate pe post de al doilea operand sursa.

Secventa initiala:

SUB R3, R6, #3
ADD R4, R3, #1 /* instructiunea care se infiltreaza */

Secventa modificata:

SUB R3, R6, #3; ADD R4, R6, #-2

❖ **MOV Reabsorption**

Acest tip de combinare implica transformarea instructiunii MOV într-o instructiune de acelasi tip cu prima.

Secventa initiala:

```
        ADD R3, R4, R5
        MOV R6, R3          /* instructiunea care se infiltreaza */
```

Secventa combinata:

```
        ADD R3, R4, R5; ADD R6, R4, R5
```

Ideea aflata la baza acestei metode este de a absorbi instructiunea MOV prin *renaming* aplicat registrului destinatie al primei instructiuni reducând astfel expansiunea codului. În cazul în care prima instructiune este una cu referire la memorie (Load sau Store) duplicarea instructiunilor poate duce la reducerea performantei datorita numarului limitat de porturi de date ale cache-ului.

Prin tehnica *merging* pot aparea doua conflicte majore. Primul este acela ca prin combinare instructiunile vor fi inserate mai degraba la mijlocul grupului de instructiuni decât la sfârsit. Al doilea se refera la faptul ca daca o instructiune este inserata în mijlocul grupului poate fi necesar ca ea sa fie redenumita. O instructiune va fi inserata în mijlocul grupului daca ea trece de câteva instructiuni din grup si va fi combinata cu alta. Întrucât prin combinare, în general, se altereaza operanzii sursa ai instructiunii care se infiltreaza, tehnica *merging* poate introduce o falsa dependenta de date (WAR, WAW) cu una din instructiunile din grup situata dupa prima instructiune aferenta tehnicii *merging*. Pentru a elimina aceasta falsa dependenta instructiunea care se infiltreaza urmeaza sa fie introdusa înaintea instructiunii care ar implica dependenta. Inserarea în mijlocul unui grup poate implica ulterior false dependente deoarece operandul destinatie poate deveni sursa pentru una din instructiunile situate în continuare spre sfârsitul grupului. Aceste false dependente pot fi eliminate prin renaming aplicat registrului destinatie a instructiunii inserate. Urmatorul exemplu exemplifica cele enuntate anterior.

LIW1: ADD R1, R2, R3; MOV R7, R8; LD R8, (R0,R5); SUB R9, R3, #4
LIW2: ADD R3, R7, R4 /* instructiunea care se infiltreaza */

Instructiunea de adunare din LIW2 se va combina cu instructiunea MOV din LIW1 creând noua instructiune ADD R3, R8, R4 si va fi necesar sa fie introdusa în fata instructiunii LD pentru a evita o falsa dependenta prin registrul R8. În plus, deoarece instructiunea SUB din LIW1 este dependenta WAR prin registrul R3 fata de instructiunea care se infiltreaza, acesta va trebui redenumit în instructiunea de adunare. Codul final va arata astfel:

LIW1: ADD R1, R2, R3; MOV R7, R8; **ADD R6, R8, R4**; LD R8, (R0,R5); SUB R9, R3, #4
LIW2: MOV R3, R6 /*instructiune introdusa datorita renaming-ului aplicat lui R3 */

Ca o concluzie, instructiunile cu care se combina instructiunile inserate nu sunt sterse sau alterate niciodata din motivul ca valoarea din registrul lor destinatie poate fi necesara altor instructiuni ulterioare. Ca rezultat, instructiunea MOV din LIW2 poate fi stearsa doar daca si când o alta instructiune care odata inserata în grup determina ca registrul R6 nu mai este necesar (teoretic a expirat durata lui de viata).

Combining – ul instructiunilor

Tehnica de *combining*, numita si colapsare statica a dependentelor de date, este identica în principiu cu tehnica *merging* în sensul depasirii problemelor legate de dependente reale de date, combinarea si schimbarea operanzilor în instructiuni pentru a permite instructiunii percolante sa-si continue drumul prin basic block. Echipa de cercetatori de la IBM (VLIW Group) condusa de Kemal Ebcioglu foloseste exemplele tehnicii "*immediate merging*" drept exemple de combining al instructiunilor. Daca tehnica "merging" restrictiona instructiunile care pot fuziona la acelea care au un numar maxim de 3 operanzi, combining-ul elimina aceasta restrictie. Pentru implementarea combining-ului sunt necesare instructiuni speciale cu 4 operanzi. La colapsarea statica a unei instructiuni care se infiltreaza rezultatul este o instructiune cu trei operanzi, totodata retinându-se si prima instructiune. În loc de o instructiune, prin colapsare sunt adaugate în scheduler doua instructiuni separate. Doar câmpul *tag* aferent fiecarei instructiuni indica faptul ca cele doua instructiuni sunt colapsate. Avantajul major al acestui aranjament este ca prima instructiune din perechea colapsata poate fi reinfiltrata ulterior sau recombinata cu o alta instructiune. Reinfiltrarea implica de catre scheduler retinerea unei perechi combinate în doua instructiuni separate. Exemplul urmator va clarifica cele enuntate anterior.

Secventa initiala:

```
MULT R7, R9, #14
ADD  R6, R7, R5
```

Secventa combinata:

MULT R7, R9, #14; **MULT R7, R9, #14; ADD R6, R7, R5**
/*instructiuni combinate logic o instuctiune */

Codul final arata astfel:

MULT R7, R9, #14; ADD R6, (R9 * #14), R5

Rezultatele obtinute prin simulare de tip “trace driven” determina o îmbunatatire a performantei procesoarelor prin folosirea tehnicii de combining de la 50 pâna la 75%.

Analiza anti-alias a referintelor la memorie

Dependentele de date nu apar doar între registri, ci si între locatii de memorie referite în instructiunile LD si ST. La fel ca celelalte dependente ele provoaca deseori degradarea performantei procesoarelor.

Pentru a face distinctie între locatiile de memorie referite de cele doua tipuri de instructiuni, HSS foloseste o tehnica numita analiza anti-alias statica a memoriei (static memory disambiguation). Pentru a decide daca o instructiune LD poate fi inserata în fata unei instructiuni ST, lucru ce se poate face în siguranta doar daca cele doua adrese difera, adresele locatiilor de memorie sunt comparate si este returnata una din valorile:

- **Diferit:** Adresele sunt întotdeauna diferite.
- **Identic:** Adresele sunt întotdeauna identice.
- **Esueaza:** Adresele nu se pot distinge.

Daca valoarea returnata este “Diferit” instructiunea LD poate fi inserata în fata instructiunii ST. De asemenea, daca valoarea returnata este “Identic” instructiunea LD poate fi înlocuita cu o instructiune MOV ca în exemplul urmator:

Secventa initiala:

```
ST (R0, R5), R6
LD R10, (R0, R5)
```

Secventa devine:

```
MOV R10, R6
ST (R0, R5), R6
```

Pe de alta parte, în cazul în care instructiunea LD este urmata de o instructiune ST, pentru ca aceasta din urma sa poata fi infiltrata în fata primeia **trebuie** ca cele doua adrese sa fie distincte întotdeauna.

Locatiile de memorie nu pot fi întotdeauna discriminate în momentul compilarii (static). Tehnica cu care se rezolva problema în acest caz se numeste anti-alias dinamic al referintelor la memorie. În acest caz instructiunile Load si Store sunt înlocuite cu cod în care comparatia celor doua adrese se face dinamic, în momentul executiei.

Exemplu:

```
     Secventa initiala:
     ST      4(R5), R8
     LD      R9, 8(R6)

     Secventa obtinuta în urma analizei anti-alias:
     ADD     R3, R5, #4     /* Calculeaza adresa pentru ST */
     ADD     R4, R6, #8     /* Calculeaza adresa pentru LD */
     LD      R9, 8(R6)      /* La adrese diferite efectuam anticipat LD */
     EQ      B1, R3, R4     /* Compara adresele pentru egalitate */
TB1  MOV     R9, R8         /* În caz de egalitate valoarea lui R9 se preia
                            direct din registru R8 */
     ST      4(R5), R8      /* Se memoreaza R8 la adresa ceruta */
```

Folosirea instructiunii gardate înlatura necesitatea prezentei a doua instructiuni de salt.

Macroexpandarea procedurilor

Tehnica de *inlining* (macroexpandare) este un mecanism prin care secventa de instructiuni din care este compusa o functie sau o procedura este duplicata si inserata în procedura apelanta în locul apelului functiei. Instructiunile de apel si revenire din procedura / functie pot fi eliminate precum si alte instructiuni care manipuleaza stiva, salveaza si restaureaza registri la intrarea si iesirea din procedura / functie.

HSS poate aplica mecanismul de macroexpandare asupra procedurilor înainte ca procesul de scheduling sa înceapa. HSS asigura câtiva parametri care controleaza tehnica *inlining* – în cazul invocarii sale de catre scheduler. Acesti parametri pot semnifica daca mecanismul se aplica procedurilor recursive sau definesc numarul maxim de basic block-uri pe care o procedura, apelata din interiorul / exteriorul unei bucle, îl poate contine pentru a putea fi macroexpandata, numarul maxim de apeluri din exteriorul unei bucle al unei proceduri, pentru macroexpandarea acesteia; numarul maxim de basic block-uri al unei proceduri apelata din interiorul unei proceduri recursive; numarul maxim de imbricari al procedurilor care pot fi macroexpandate.

Înainte de a sti daca mecanismul de inlining va fi invocat sau nu se înregistreaza informatiile privitoare atât la basic block-uri cât si la proceduri. Se identifica si se înregistreaza datele privind procedurile

recursive. La invocarea mecanismului de macroexpandare se disting trei cazuri majore:

- Apelul procedurii / functiei se face din interiorul unei bucle;
- Apelul procedurii / functiei se face din interiorul unei proceduri recursive dar în afara buclei;
- Apelul procedurii / functiei se face din exteriorul unei bucle.

Imediat ce procedura macroexpandata a fost inserata în codul programului, toate etichetele din procedura respectiva sunt renumite si toate tintele branch-urilor sunt actualizate. Orice instructiuni redundante de intrare sau iesire, în sau din procedura, sunt eliminate. Printre aceste instructiuni se afla si cele de load / store care salveaza registrii la intrarea în procedura respectiv îi restaureaza la iesire, deoarece valorile din registrii respectivi nu mai sunt folosite de procedura apelanta. Apelurile BSR, precum si ultima instructiune din procedura – MOV PC, RA – nu mai sunt necesare si sunt eliminate.

Prezentam, în continuare, un exemplu edificator al mecanismului de inlining. Consideram doua proceduri Proc1 si Proc2 înainte si dupa macroexpandare.

Proc1:		**Proc2:**	
L1:	SUB SP, SP, #256		SUB SP, SP, #256
	ST 8(SP), R16		ST 8(SP), R16
L2:	LD R16, 8(SP)		
	ADD SP, SP, #256		
	MOVPC, RA		BSR RA, Proc1
		L4:	MOV R5, R16
			LD R16, 8(SP)
			ADD SP, SP, #256
			MOV PC, RA

Dupa macroexpandare procedura Proc1 nu se schimba însa Proc2 va contine si instructiunile lui Proc1, astfel:

```
Proc1:                          Proc2:
L1:     SUB  SP, SP, #256               SUB  SP, SP, #256
        ST   8(SP), R16                 ST   8(SP), R16

L2:     LD   R16, 8(SP)         Q1:     SUB  SP, SP, #256
        ADD  SP, SP, #256               ST   8(SP), R16
        MOV  PC, RA
                                        Proc1 macroexpandata

                                Q2:     LD   R16, 8(SP)
                                        ADD  SP, SP, #256

                                L4:     MOV  R5, R16
                                        LD   R16, 8(SP)
                                        ADD  SP, SP, #256
                                        MOV  PC, RA
```

Odata aplicat procedurilor mecanismul de inlining noile benchmark-uri sunt puse la dispozitia scheduler-ului.

Algoritmul HSS

Algoritmul HSS are un dublu obiectiv. Primul consta în aplicarea tehnicii *software pipelining* la bucle de o complexitate arbitrara. Al doilea obiectiv este de a reduce expansiunea codului prin evitarea miscarii neproductive a codului de o parte si de alta a marginii buclei. În continuare se prezinta sintetic acest algoritm.

Etapa I

Reorganizarea buclei cu infiltrarea instructiunilor este permisa doar în interiorul buclei.

Etapa II

Repeta {

Infiltrarea fiecarei instructiuni din primul grup de instructiuni al buclei în jurul marginii de reluare a buclei.

1. Se determina existenta dependentelor dintre instructiunea care se infiltreaza si alte instructiuni din grupurile finale de instructiuni aflate în interiorul buclei. De observat ca

pot apare antidependente. (În cazul instructiunilor de ramificatie este suficient ca lantul dependentelor de date sa ajunga la un branch).

2. Se verifica daca lantul dependentelor de date poate fi colapsat folosind fie tehnica de *merging* fie cea de *combining* aplicata instructiunilor.

Daca cel putin o instructiune si-a schimbat pozitia, bucla se recompacteaza si se reia procesul de infiltrare al instructiunilor doar în interiorul buclei.

}*pâna când* (nici o instructiune nu mai poate fi mutata)

Pentru clarificarea functionarii algoritmului HSS prezentam etapele de transformare a unui basic block prin scheduling, subliniind reducerea numarului de iteratii al buclei. Presupunem ca toate instructiunile au latenta unitara si ca *branch delay slot*-ul este 0.

Codul original:

Loop:

tact=1	LD	R16, (R1)
tact=2	ADD	R16, R16, #17
tact=3	ST	(R3), R16
tact=4	ADD	R1, R1, #4
tact=5	ADD	R3, R3, #4
tact=6	SUB	R2, R2, #1
tact=7	NE	B1, R2, #0
tact=8	BT	B1, Loop

Bucla initiala necesita 8 cicli pentru a fi executata. Pe parcursul primei treceri fiecare instructiune începând din marginea superioara a buclei se infiltreaza si promoveaza în sus în structura de cod cât se poate de mult.

Loop:
tact=1 LD R16, (R1); ADD R1, R1, #4; ADD R4, R3, #4; SUB R2, R2, #1;
tact=2 ADD R16, R16, #17; NE B1, R2, #0;
tact=3 ST (R3), R16; MOV R3, R4; BT B1, Loop

Registrul destinatie al celei de-a treia instructiuni ADD este redenumit din R3 în R4. Fiecare bucla se executa acum doar în 3 cicli în loc de 8.

Instructiunile din primul grup de instructiuni promoveaza apoi în jurul marginii de reluare a buclei. În cazul aparitiei dependentelor de date care nu pot fi înlaturate dintre instructiunea care se infiltreaza si alte instructiuni, procesul de promovare se întrerupe. În timpul primei infiltrari în jurul marginii buclei se creeaza o zona de cod *prolog*. Fiecare instructiune care se misca în jurul acestei margini poate promova de asemenea si în zona de cod preludiu. Bucla este reorganizata si procesul de promovare al instructiunilor se reia întâi în interiorul buclei, apoi si în jurul ei. Procesul continua până când nici o instructiune nu mai poate fi mutata si codul arata astfel:

Codul prolog:

tact=1 LD R17, (R1); SUB R2, R2, #1; ADD R1, R1, #4;
tact=2 NE B1, R2, #0; SUB R2, R2, #1; ADD R16, R17, #17;
tact=2 LD R17, (R1); ADD R1, R1, #4

Loop1:

tact=3 SUB R2, R2, #1; ST (R3), R16; ADD R3, R3, #4;
tact=3 ADD R3, R3, #4; ADD R16, R17, #17; LD R17, (R1);
tact=3 ADD R1, R1, #4; BT B1, Loop1; NE B1, R2, #0;

Zona de cod prolog cuprinde iteratii partiale ale buclei în diferite faze de executie. Corpul buclei completeaza iteratiile începute în codul preludiu. Bucla a fost restrânsa la o singura iteratie, executia durând un ciclu.

Concluzii

Lucrarea de fata se constituie într-o introducere în scheduling si considera schedulerul HSS un exponent modern al acestei tehnici. Desi efortul autorilor a fost de a realiza o documentatie cât mai completa, multe detalii si caracteristici au fost omise pentru a forma cititorului o impresie precisa si clara despre scheduler si scheduling.

5. PROCESORUL IA-64: ÎNTRE EVOLUTIE SI REVOLUTIE

Arhitectura Intel IA-64 pe 64 de biti (având numele de cod *Merced*) a fost proiectata de catre Intel în colaborare cu cercetatorii de la Hewlett Packard dar si cu anumite grupuri de cercetare academice precum cel de la Universitatea din Illinois (compilatorul IMPACT), ca reprezentând un pas (r)evolutionar pe linia viitoarelor microprocesoare (de dupa 1999) comerciale de uz general, prin exploatarea agresiva a paralelismului la nivel de instructiuni printr-o sinergie hardware-software numita EPIC ("*Explicitly Paralell Instruction Computing*"). Practic IA-64 va înlocui curând si pe plan comercial deja clasicul standard Pentium, cu unul nou, practic aproape necunoscut momentan având în vedere ca prima documentatie tehnica cuprinzînd cca. o mie de pagini a fost data publicitatii abia în mai 1999 [34]. În vederea extragerii unui grad ILP ("Instruction Level Parallelism") maximal, procesorul IA-64 (Intel Architecture) - ce va fi cunoscut sub prima forma comerciala ce va fi implementata în tehnologie de 0.18 microni la 800 MHz sub numele de *Itanium* - include caracteristici arhitecturale moderne precum: executie speculativa si predicativa a instructiunilor, seturi extinse de registri interni, predictor avansat de branch-uri, optimizatoare de cod etc. Adresarea memoriei se face pe 64 biti într-un spatiu virtual urias. Desigur ca IA-64 respecta o compatibilitate binara perfecta cu arhitecturile Intel pe 32 biti anterioare (Pentium, Pentium Pro, Pentium II, Pentium III), putându-se deci rula actualele aplicatii soft pe 32 biti pe noile platforme de operare pe 64 biti. De altfel, arhitectura IA-64 permite atât implementarea unor sisteme de operare pe 32 biti în modurile de lucru protejat, real si virtual 8086 (Pentium) cât si a unor sisteme de operare pe 64 biti care însa pot rula mai vechile (actualele !) aplicatii pe 32 de biti în oricare dintre cele 3 moduri de lucru cunoscute. Asadar în orice moment, procesorul acesta poate rula atât instructiunile Pentium ISA-32 ("Instruction Set Arhitecture") cât si setul

nou ISA-64. Exista implementate 2 instructiuni de salt neconditionat dedicate (având mnemonicile *jmpe* si *br.ia*) care dau controlul programului spre o instructiune ISA-64 respectiv ISA-32 si totodata modifica în mod corespunzator setul curent de instructiuni utilizate (ISA). În continuare se vor prezenta succint si fatalmente incomplet, doar **câteva dintre caracteristicile arhitecturale novatoare** ale IA-64, unele implementate în premiera în sfera comerciala (altfel cunoscute si investigate de mult timp în mediile de cercetare, în special academice).

Arhitectura generala: cîteva aspecte

Pe scurt, arhitectura registrilor program ar fi urmatoarea:

- 128 registri generali pe 64 biti (32 globali si 96 locali, utilizabili in ferestre de catre diversele programe). În modul de lucru pe 32 biti IA-32 (compatibil Pentium), parte sau portiuni din acesti registri generali îndeplinesc rolul cunoscutilor registri ai acestei arhitecturi. Astfel de exemplu registrul GR8 devine acumulatorul extins EAX iar registrul GR17(15:0) devine registrul selector CS din cadrul modului de lucru protejat IA-32.
- 128 registri FPP (*Flotant Point Processor*) pe 82 de biti (primii 32 sunt generali, ceilalti sunt dedicati redenumirii statice în vederea accelerarii procesarii buclelor de program).
- 64 registri de garda (registri predicat) pe 1 bit, utilizati pentru executia conditionata a instructiunilor (vezi în continuare).
- 8 registri destinati instructiunilor de branch, pe 64 de biti. Sunt utilizati pentru a specifica adresele destinatie în cazul salturilor indirecte (inclusiv cele de tip call/return).
- un numarator de adrese program numit – clasic în tehnologia Intel – Instruction Pointer (IP).
- un registru numit CFM ("Current Frame Marker") care descrie starea ferestrei curente de registri locali (marime fereastra, numar registri locali/de iesire, adrese de baza pentru registrii utilizati la redenumire [8, 33, 34]).
- un numar de asa-numiti registri de aplicatie incluzând registrii de date dedicati unor scopuri speciale precum si registrii de control ai aplicatiilor.

IA-64 proceseaza 6 tipuri distincte de instructiuni (întregi ALU, întregi non-ALU, cu referire la memorie, flotante, ramificatii si extinse). Exista implementate 4 unitati de executie si anume:

- unitatea I dedicata instructiunilor întregi si extinse

- unitatea M dedicata instructiunilor cu referire la memorie dar si unor instructiuni de tip întreg ALU
- unitatea F dedicata instructiunilor în virgula mobila (FPP)
- unitatea B (Branch) dedicata instructiunilor de ramificatie program

Instructiunile IA-64 sunt procesate pipeline-izat în 4 faze distincte si anume:

- faza de aducere a instructiunii din cache-ul de instructiuni sau din memoria principala
- faza de citire stare arhitecturala, daca e necesara
- faza de executie propriu-zisa a instructiunii
- faza de actualizare a contextului arhitectural, daca e necesara ("update")

Astfel IA-64 permite executia mai multor instructiuni – masina independente simultan, fapt facilitat prin implementarea unor **unitati de executie multiple** , seturi multiple de registri generali, optimizator de cod agresiv (scheduler integrat în compilatoare si care grupeaza instructiunile independente din programul obiect în grupuri de instructiuni primitive multiple) etc. Arhitectura permite transferuri de informatie între compilator si procesorul hardware în scopul minimizarii efectelor defavorabile aferente miss – predictiei ramificatiilor, miss-urilor în cache, instructiunilor load/store etc. Astfel de exemplu, compilatorul poate transmite prin anumite câmpuri binare din codul instructiunii, informatii deosebit de utile legate de predictia branch-urilor (strategie de predictie statica sau dinamica prin predictorul hardware etc.) [33].

Schedulerul "împacheteaza" câte 3 instructiuni primitive independente într-o asa numita instructiune multipla ("bundle") dupa binecunoscutul principiu care sta la baza arhitecturilor VLIW ("Very Long Instruction Word"). Formatul unei asemenea instructiuni multiple este prezentat mai jos:

Slot instr. 1	Slot instr. 2	Slot instr. 3	Template
41 biti	41 biti	41 biti	5 biti

Câmpul "template" are 2 roluri distincte:

- codifica unitatea de executie aferenta fiecarei instructiuni primitive din instructiunea multipla, realizând astfel o rutare statica a instructiunilor spre unitatile de executie cu beneficii majore asupra reducerii complexitatii hardware a procesorului

- indica asa-numite “stopuri arhitecturale”; un atfel de “stop” aferent unui slot informeaza structura hardware ca una sau mai multe instructiuni anterioare sunt dependente de date fata de una sau mai multe instructiuni situate dupa cea marcata cu “stop”. Prin urmare, logica de control va opri fluxul de instructiuni pe slotul respectiv pâna la rezolvarea dependentei în cauza. Cu alte cuvinte, structura hardware este asigurata, de catre schedulerul (reorganizatorul - optimizator) software, ca toate instructiunile primitive cuprinse între 2 astfel de “stopuri” succesive sunt independente (RAW – Read After Write sau WAW – Write After Write) si prin urmare, ar putea fi procesate în paralel daca resursele hardware disponibile o permit. Se rezolva astfel în mod elegant o redundanta operationala caracteristica din pacate multor procesoare superscalare actuale, prin reluarea de catre hardware a stabilirii dependentelor de date, operatie efectuata anterior de catre schedulerul software. Iata deci esenta conceptului de “paralelism explicit” (EPIC) ce caracterizeaza arhitectura IA-64 ca fiind **un hibrid interesant între o structura superscalara simplificata si respectiv una tip VLIW.**

Executia speculativa a instructiunilor

O alta caracteristica deosebit de moderna implementata în arhitectura IA-64 (microprocesorul *Itanium*) consta în **executia speculativa** a instructiunilor. Exista 2 tipuri de speculatii: **de control** (când o instructiune aflata în program dupa un salt conditionat se executa înaintea acestuia cu influenta benefica asupra timpului de executie) si respectiv **de date** (când o instructiune tip “load” situata în program dupa o instructiune tip “store” se executa înaintea acesteia cu beneficii asupra timpului de executie datorate mascarii latentei instructiunii de încarcare). Pentru a exemplifica în mod concret o speculatie de control pe IA-64 se considera secventa de program:

```
if (a > b)
    load (ld_adr1,target 1)
else
    load (ld_adr2,target 2)
```

Schedulerul va rescrie aceasta secventa utilizând instructiuni “load” de tip speculativ (“sload”), ca mai jos:

```
sload (ld_adr1,target1)
sload (ld_adr2,target2)
if (a>b)
```

```
    scheck (target1, recovery_adr1)
else
    scheck (target2, recovery_adr2)
```

În primul rând se observa ca instructiunile de încarcare se executa speculativ, înainte de determinarea conditiei de salt (a>b), conducând astfel la grabirea procesarii. Desigur în acest caz sunt necesare corectii în situatia în care una dintre cele 2 încarcari speculative genereaza o exceptie (de exemplu o exceptie de tip "page fault"). Pentru prevenirea unor asemenea evenimente nedorite se verifica prin instructiuni corespunzatoare ("scheck") daca una dintre cele 2 instructiuni speculative au generat vreo exceptie si în caz afirmativ se pointeaza la o rutina de restabilire a contextului programului (recovery_adr) în vederea asigurarii unui mecanism precis de exceptii.

Ca exemplu de speculatie de date se considera secventa:

```
store (st_adr,date)        ; scriere în memoria de date
load (ld_adr,target)       ; citire din memoria de date
use (target)               ; utilizare variabila
```

Cum în acest caz mecanisme de analiza statica antialias (st_ad = ld_adr ?) nu sunt practic posibile, executia "out – of – order" a acestor instructiuni este posibila numai prin utilizarea unor mecanisme speculative. Si în cazul secventei rescrise speculativ sunt necesare corectii în cazul unei exceptii produse prin executia speculativa a instructiunii "load" ori în cazul în care se dovedeste ca ld_adr = st_adr. Într-o astfel de situatie se da controlul unei rutine de restabilire a contextului în conformitate cu logica initiala de program, rutina care începe la adresa "recovery_adr" (se implementeaza deci un mecanism de exceptii precise). Secventa optimizata ar fi în acest caz urmatoarea:

```
aload (ld_adr,target)
store (st_adr,data)
acheck(target,recovery_adr) ;verificare eveniment nedorit
use (target)
```

Principiile executiei predicative

O alta caracteristica arhitecturala importanta inclusa în procesoarele IA-64 consta în **executia predicativa** a instructiunilor [33, 34]. În aceasta filozofie, o instructiune se executa efectiv sau nu ("nop") în functie de valoarea de adevar a unei variabile booleene de garda sau a mai multor

astfel de variabile conjugate prin SI logic. De exemplu instructiunea P5 R6=(R7)+(R9) executa adunarea daca variabila de garda P5=”true”, în caz contrar neexecutându-se practic nimic (“nop”). Desigur ca aceste variabile de garda booleene se seteaza/reseteaza ca urmare a actiunii unor instructiuni (predicate de ordinul întâi în general). Ca exemplu în acest sens, instructiunea P=compare (a>b), face P=”true” daca a>b si respectiv P=”false” daca a≤b. Majoritatea instructiunilor IA-64 sunt executabile conditionat. Prin utilizarea unor transformari ale programului obiect bazat pe instructiuni gardate se elimina circa 25% - 30% dintre instructiunile de salt, marindu-se astfel lungimea medie a unitatilor secventiale de program (“basic – block”-uri) cu influente favorabile asupra procesului de scheduling [8, 30, 33]. Este adevarat însa ca, dependentele datorate instructiunilor de salt canditionat (branch) se vor regasi acum sub forma unor noi dependente de date prin variabilele de garda (RAW – dependente tip “Read after Write”) si care conduc la secventialitatea executiei programului. Se prezinta mai jos transformarea unei secvente de program prin predicare cu eliminarea saltului:

```
if (a>b)              ; P6=compare (a>b)
    c=c+1             ; TP6 c=c+1;
else
    d=d*e+5           ; FP6 d=d*e+5;
```

Se obsera ca dependenta de ramificatie (a>b) s-a transformat acum în dependenta RAW prin variabila de garda P6. Un beneficiu suplimentar consta în paralelizarea celor 2 instructiuni gardate si exclusive din punct de vedere al executiei lor (TP6, FP6).

Preluat din [34], în continuare se prezinta un exemplu sugestiv în legatura cu avantajele/dezavantajele executiei predicative a instructiunilor în cadrul IA-64. Se considera secventa “if – then – else” ca mai jos :

```
if (r4)
    r3 = r2 + r1      ; 2 cicli
else
    r3 = r2 * r1
utilizare r3 ;        ; 18 cicli
```

S-a considerat deci ca ramificatia “if” are latenta de 2 cicli procesor iar ramificatia “else” de 18 cicli procesor. Secventa va fi compilata prin instructiuni gardate, eliminându-se instructiunile de ramificatie, ca mai jos:

```
     cmpne p1,p2 = r4,r0      ;0/0 compara pe diferit
(p1) add r3 =r2,r1            ;1/1 adunare întregi
(p2) setf f1=r1               ;1/1 conversie întreg – flotant
(p2) setf f2=r2               ;1/1
(p2) xma.l f3=f1,f2           ;9/2 înmultire f1xf2 (flotant)
(p2) getf r3=f3               ;15/3 conversie flotant – întreg
(p2) utilizare r3             ;17/4
```

În comentariu, imediat dupa semnul ";", sunt scrise 2 cifre: prima semifica numarul ciclului în care instructiunea respectiva va fi lansata în executie daca variabila de garda p1 = 1 ("true") iar a 2–a acelasi lucru în cazul contrar p1 = 0 (implicit p2 = 1). Considerând acum ca "setf" dureaza 8 cicli, "getf" 2 cicli, "xma" 6 cicli si ca o predictie incorecta a branch-ului costa procesorul 10 cicli (pt. restaurarea starii), se pot analiza 2 cazuri complementare sugestive.

Cazul I

Se presupune ca ramura "if" se executa 70% din timp iar acuratetea predictiei branch-ului din codul initial (nepredicativ) este 90%. Timpul de executie al secventei initiale este:

(2 cicli x 70%) + (18 cicli x 30%) + (10 cicli x 10%) = 7.8 cicli

Timpul de executie al secventei compilate prin predicare este:

(5 cicli x 70%) + (18 cicli x 30%) = 8.9 cicli

În acest caz executia predicativa este neeficienta.

Cazul II

Se presupune ca ramura "if" se executa 30% din timp si ca acuratetea predictiei branch-ului este acum de doar 70%. Timpul de executie al secventei initiale este:

(2 cicli x 30%) + (18 cicli x 70%(+ (10 cicli x 30%) = 16.2 cicli

Timpul de executie al secventei compilate prin predicare este:

(5 cicli x 30%) + (18 cicli x 70%) = 14.1 cicli

În acest al 2-lea caz executia predicativa este mai eficienta, micsorând timpul mediu de executie cu mai mult de 2 cicli.

Register Windows sau "chemarea strabunilor"

O alta caracteristica interesanta, desi utilizata de catre pionierii procesoarelor RISC ("Reduced Instruction Set Computing") înca din 1980 în cadrul microprocesorului Berkeley I RISC (Prof. David Patterson, Universitatea Berkeley, SUA), consta în lucrul **"în ferestre de registre"** ("register windows",vezi [35, 33, 3]). Aceasta tehnica este în legatura cu minimizarea timpului de intrare/revenire într-o/dintr-o procedura. Astfel, se evita atunci când este posibil salvarea în stiva a setului de registri locali ai programului apelant în vederea reutilizarii sale în cadrul procedurii apelate. Acest lucru se realizeaza exclusiv in modul ISA-64 prin alocarea unui nou set (ferestre) de registri locali procedurii apelate. Mai precis, IA-64 contine 32 de registri generali (GR0 – GR31), utilizabili de catre toate procedurile (globali) si respectiv 96 de registri utilizabili în ferestre dinamice (locali), alocabili deci prin software diferitelor proceduri (*alloc*). La rându-i, o fereastra de registri locali alocata procedurii în curs, contine 2 zone distincte: o zona care contine registrii locali de lucru si parametrii de intrare generati de catre programul apelant si respectiv o zona de registri care memoreaza parametrii de iesire (rezultate) ce vor fi utilizati la revenire de catre programul apelant. La revenirea din procedura (*return*) se comuta automat pe fereastra anterioara de registri locali. Desigur salvarea/restaurarea registrilor locali în/din stiva apare ca necesara numai în cazul unor depasiri ale setului secundar de 96 de registri utilizabili în ferestre dinamice. Este interesanta implementarea acestei tehnici în cadrul modernei arhitecturi IA-64 întrucât desi utilizata înca de la începuturile generatiei a 2-a arhitecturale de microprocesoare ("pipeline" RISC), ea nu a fost foarte agreata ulterior (exceptie face famila de microprocesoare SPARC a companei SUN), din cauza cresterii latentei caii critice de date a microprocesoarelor (timpul maxim necesar celei mai lungi operatii interne încadrabile într-un ciclu) si deci în consecinta, reducerii frecventei de tact a acestora. Probabil ca Intel Co. renunta deliberat la suprematia frecventelor de tact (aici microprocesoarele din familia Alpha 21264 - Compaq de exemplu sunt mereu cu un pas înainte) mizând pe cresterea gradului mediu de ILP extras prin metode arhitecturale de sinergism hardware-software precum cele descrise succint pâna acum. În fond, Intel stie prea bine ca performanta se obtine actualmente preponderent prin arhitectura (cca. 65%) si abia mai apoi prin tehnologie (cca. 35%).

Optimizarea buclelor de program

Arhitectura IA-64 se bazeaza esentialmente pe **forta compilatorului (schedulerului)** care exploateaza în mod static paralelismul la nivel de instructiuni din codul obiect. Optimizarea buclelor de program este esentiala întrucît, dupa cum prea bine se stie, cca. 90% din timp se executa cca.10% din program, iar acesta fractiune consta probabil cu precadere în bucle. Dintre tehnicile soft de paralelizare a instructiunilor aferente buclelor de program, IA-64 utilizeaza tehnica "modulo scheduling" (M. Lam, B. Rau) care în esenta transforma bucla initiala astfel încât sa se permita executia simultana a unor iteratii diferite, prin pipeline-izarea software a acestora.

Pentru a da concretete acestor aspecte se va prezenta în continuare un exemplu de optimizare a unei bucle simple de program prin utilizarea tehnicii de paralelizare a unor iteratii diferite ("software pipelining" – tehnica atribuita cercetatoarei americane Monica Lam).

Se considera bucla de program IA-64:

```
L1:
    ld r4 = [r5],4 ;încarcare din memorie cu postincrementarea adresei (+4)
    add r7= r4,r9
    st [r6]= r7,4 ;memorare r7 cu postincrementarea adresei (r6 + 4 -> r6)
    br.cloop L1 ;loop
```

Instructiunea de buclare inspecteaza un registru special în care s-a încarcat initial numarul de iteratii aferent buclei, si daca acesta are un continut pozitiv, se decrementeaza si se face saltul.

De remarcat ca toate instructiunile buclei sunt dependente, ceea ce conduce la o executie fatalmente seriala a acestora în concordanta cu implacabila (?) lege a lui Eugene Amdahl. Având în vedere avatariile instructiunii de salt în structurile pipeline si superscalare, se poate considera ca executia acestei secvente de program este una ineficienta, lenta. Schedulerul ar putea transforma aceasta bucla, în urma aplicarii tehnicii "software pipelining" (TSP), ca mai jos:

	ld	r4 = [r5],4	;iteratia 1	
	add	r7 = r4,r9	;iteratia 1	**PROLOG**
	ld	r4 = [r5],4	;iteratia 2	
L1:	st	[r6]=r7,4	;iteratia k∈[1,n-2]	
	add	r7 = r4,r9	;iteratia k+1	**NUCLEU**
	ld	r4 = [r5],4	;iteratia k+2	
	br.cloop L1			

```
st      [r6] = r7,4     ;iteratia n-1
add     r7 = r4, r9     ;iteratia n            EPILOG
st      [r6] = r7,4     ;iteratia n (ultima)
```

Esenta TSP rezulta imediat analizând secventa **NUCLEU** care pipeline-izeaza 3 instructiuni apartinând unor 3 iteratii succesive ale buclei de program. Prin aceasta, cele 3 instructiuni ale buclei **NUCLEU** s-ar putea executa în paralel daca resursele hardware o permit. Secventele **PROLOG/EPILOG** nu fac decât sa "ajusteze" din punct de vedere semantic implicatiile buclei nucleu .

Pentru a sugera câstigul de performanta obtinut în urma aplicarii TSP, se va rescrie secventa anterioara grupând pe acelasi rând instructiunile paralelizabile, deci executabile în acelasi ciclu masina. În paranteze se va scrie numarul iteratiei careia îi apartine instructiunea respectiva.

```
        ld (1)   ld (2)               ;primul ciclu
        add (1)                       ;al 2-lea ciclu etc.
L1:     st (k)   add (k+1)    ld (k+2)
        br.cloop L1
        st (n-1) add (n)
        st (n)
```

Un alt câstig important al metodei TSP consta în faptul ca expansiunea codului optimizat este moderata si datorata exclusiv secventelor de prolog/epilog. Nu acelasi lucru se poate afirma despre alte metode de optimizare a buclelor de program (exemplu "loop unrolling" [8]), care determina expansiuni ale codului de pâna la 200% cu repercursiuni negative asupra vitezei de procesare. În mod oarecum ironic, optimizarea buclelor de program urmareste reducerea timpului de executie dar, prin expansiunea de cod implicata, creste rata de miss în cache-uri ceea ce determina cresterea timpului de executie !

Aceasta executie concurenta a unor (portiuni din) iteratii diferite , necesita frecvent redenumiri ale registrilor utilizati în vederea eliminarii dependentelor de date între acestia. IA-64 permite ca fiecare iteratie sa utilizeze propriul set de registri, evitând astfel necesitatile de desfasurare a buclelor ("loop unrolling"). De asemenea se mentioneaza ca IA-64 are inclus un puternic procesor FPP (Flotant Point Processor) în virgula mobila, înzestrat inclusiv cu facilitati 3D si un procesor multimedia compatibil semantic cu tehnologia Intel MMX si SIMD ("*Single Instruction Multiple Data*" – model vectorial).

O alta tehnica foarte cunoscuta de optimizare a buclelor de program este asa numita “Loop Unrolling” (LU) care se bazeaza pe desfasurarea buclei de un numar de ori si apoi optimizarea acesteia [8, 33]. În cadrul diferitelor copii de iteratii concatenate se redenumesc anumiti registri în vederea eliminarii dependentelor de date de tip WAR sau WAW. Pentru exemplificarea tehnicii LU se considera bucla anterioara:

```
L1:
    ld r4 = [r5],4      ;ciclul 0
    add r7= r4,r9       ;ciclul 2, s-a presupus aici latenta instr. “ld” de 2 cicli
    st [r6]= r7,4       ;ciclul 3
    br.cloop L1         ;ciclul 3
```

Iata ce devine aceasta bucla simpla dupa ce este desfasurata de catre compilator de 4 ori (s-a presupus ca numarul de iteratii este multiplu de 4 si ca memoria cache de date detine 2 porturi de acces) iar apoi optimizata în vederea unei procesari optimale:

```
    add r15 = 4,r5
    add r25 = 8,r5
    add r35 = 12,r5
    add r16 = 4,r6
    add r26 = 8,r6
    add r36 = 12,r6
L1:
    ld r4 = [r5],16         ; ciclul 0
    ld r14 = [r15],16       ; ciclul 0
    ld r24 = [r25],16       ; ciclul 1
    ld r34 = [r35],16       ; ciclul 1
    add r7 = r4,r9          ; ciclul 2
    add r17 = r14,r9        ; ciclul 2
    st [r6] = r7,16         ; ciclul 3
    st [r16] = r17,16       ; ciclul 3
    add r27 = r24,r9        ; ciclul 3
    add r37 = r34,r9        ; ciclul 3
    st [r26] = r27,16       ; ciclul 4
    st [r36] = r37,16       ; ciclul 4
    br.cloop L1             ; ciclul 4
```

Preambulul si redenumirile din cadrul buclei desfasurate sunt necesare eliminarii conflictelor de nume. Cu exceptia ciclului 2 în fiecare ciclu sunt utilizate din plin cele 2 porturi ale memoriei cache de date. Performanta obtinuta este de 4 iteratii în 5 cicli fata de o iteratie în 4 cicli cât obtinea bucla initiala, o îmbunatatire absolut remarcabila. Desigur, dupa cum se poate observa în virtutea unui necrutator dar etern compromis performanta – cost, lungimea buclei optimizate creste cu repercursiuni defavorabile evidente dar acceptabile totusi având în vedere cresterea vitezei de executie.

6. ARHITECTURA MICROPROCESOARELOR, ÎNCOTRO ?

Din punct de vedere arhitectural se considera ca pâna la ora actuala au existat 3 generatii de (micro)procesoare de succes comercial dupa cum urmeaza:

- generatia I caracterizata în principal prin executia secventiala a fazelor (ciclilor masina) aferente instructiunilor- masina. Pionierii acestei generatii sunt desigur inventatorii calculatorului numeric, inginerii *Eckert* si *Mauchly*, alaturi de cel care ulterior a teoretizat si a îmbogatit conceptul, în persoana marelui om de stiinta american *John von Neumann*.
- generatia a II-a de procesoare, exploata paralelismul temporal aferent instructiunilor masina prin suprapunerea fazelor (pipeline). Primul reprezentant comercial a fost sistemul CDC-6600 (1964) proiectat de catre cel mai mare creator de calculatoare de înalta performanta si totodata unul dintre pionierii supercalculatoarelor, *Seymour Cray*. În anii '80, (micro)procesoarele RISC scalare au reprezentat aceasta generatie (*J. Cocke* de la IBM si *D. Patterson* de la Univ. Berkeley fiind doar doi dintre pionierii promotori ai acestor idei).
- generatia a III-a, cea curenta, este caracterizata de procesarea mai multor instructiuni independente simultan prin exploatarea unui paralelism spatial la nivelul diverselor unitati functionale de procesare. Executia instructiunilor se face Out of Order, utilizând deci tehnici de reorganizare (dinamica sau statica) a instructiunilor în vederea minimizarii timpului global de executie. Pionierul acestei generatii a fost sistemul anilor '60 IBM-360/91 (printre proiectanti *Anderson, Sparacio, Tomasulo, Goldschmidt, Earle,* etc.). La ora actuala generatia aceasta este reprezentata prin microprocesoarele superscalare, VLIW, etc.

De câtiva ani, în laboratoarele de cercetare (în special cele academice!) se întrezaresc câteva solutii privind caracteristicile majore ale urmatoarei decade, generatia a IV-a, pe care le vom analiza succint si fatalmente incomplet, în continuare.

În ultimii ani, procesul de proiectare al procesoarelor s-a modificat radical. Astazi, accentul principal nu se mai pune pe implementarea hardware, ci pe proiectarea arhitecturii. Se porneste de la o arhitectura de baza, care este modificata si îmbunatatita dinamic, prin simulari laborioase pe *benchmark*-uri reprezentative (Stanford, SPEC '92, '95, etc., pentru procesoarele de uz general). De exemplu, proiectantii firmei Intel, pentru procesorul Intel Pentium Pro (P6), au pornit de la o structura initiala care continea un pipeline cu 10 nivele, decodificator cu 4 instructiuni / ciclu, cache-uri separate pe instructiuni si date de capacitate 32Ko fiecare si un total de 10 milioane tranzistori. Comportarea fiecarei componente a arhitecturii (efectul capacitatii primului nivel (L1) cache, numarul de nivele în pipeline, comportarea logicii de predictie a salturilor, numarul de unitati functionale, etc.) a fost simulata soft prin rularea a aproximativ 200 benchmark-uri, cu peste 2 miliarde de instructiuni! Rezultatul final a impus un procesor cu un pipeline pe 14 nivele, 3 instructiuni decodificate în fiecare ciclu, 8Ko L1 cache de date si 8Ko L1 cache de instructiuni, cu un total de aproximativ doar 5.5 milioane tranzistoare integrate.

Costul proiectarii este relativ mare si include în principal elaborarea unei arhitecturi dinamice, scrierea unui compilator, de C in general, pe arhitectura respectiva, scheduler (optimizator) pentru codul obiect, simulator puternic parametrizabil si complex, programe de interpretare a rezultatelor. De exemplu, microprocesorul MIPS-4000 s-a creat prin efortul a 30 de ingineri timp de 3 ani. Costul cercetarii-proiectarii a fost de 30 milioane dolari, iar cel al fabricarii efective de numai 10 milioane dolari. Numai pentru simulare si evaluare s-au consumat circa 50.000 ore de procesare pe masini având performante de 20 MIPS [38].

Oricum, arhitecturile cu executii multiple si pipeline-izate ale instructiunilor (superscalare, VLIW) dau deja anumite semne de "oboseala", limitarile fiind atât de ordin tehnologic cât si arhitectural [51, 52, 40]. Caracteristicile arhitecturale complexe implica tehnologii tot mai sofisticate, înca nedisponibile. Pe de alta parte, performantele lor cresc asimptotic pe actualele paradigme arhitecturale. Totusi, schimbari fundamentale sunt mai greu de acceptat în viitorul apropiat, în primul rând datorita compilatoarelor optimizate, având drept scop exploatarea mai pronuntata paralelismului la nivel de instructiuni, deoarece acestea sunt deosebit de complexe si puternic dependente de caracteristicile hardware.

Exista deja opinii care arata ca arhitecturile superscalare si VLIW contin limitari fundamentale si care ar trebui analizate si eventual eliminate. Dintre aceste limitari amintim doar conflictele la resurse, datorate în principal centralizarii acestora. O idee interesanta bazata pe descentralizarea resurselor se prezinta în [37] si are în vedere implementarea mai multor asa numite "Instruction Windows" (IW)- un fel de buffere de prefetch multiple în locul unuia singur si respectiv pe conceptul de multithreading. Lansarea în executie a instructiunilor se face pe baza determinarii celor independente din fiecare IW. Desigur ca trebuie determinate si dependentele inter- IW-uri. Ideea principala consta în executia paralela a mai multor secvente de program aflate în IW- uri diferite, bazat pe mai multe unitati functionale (multithreading). Astfel de exemplu, 2 iteratii succesive aferente unei bucle de program pot fi procesate în paralel daca sunt memorate în IW- uri distincte. O asemenea idee faciliteaza implementarea conceptelor de expandabilitate si scalabilitate, deosebit de utile în dezvoltarea viitoare a arhitecturii.

În esenta, un procesor cu executii multiple ale instructiunilor este compus din 2 mecanisme decuplate: mecanismul de aducere (fetch) a instructiunilor pe post de producator si respectiv mecanismul de executie a instructiunilor pe post de consumator. Separarea între cele 2 mecanisme (arhitectura decuplata) se face prin bufferele de instructiuni si statiile de rezervare, ca în figura 6.1. Instructiunile de ramificatie si predictoarele hardware aferente actioneaza printr-un mecanism de reactie între consumator si producator. Astfel, în cazul unei predictii eronate, bufferul de prefetch trebuie sa fie golit macar partial iar adresa de acces la cache-ul de instructiuni trebuie si ea modificata în concordanta cu adresa la care se face saltul.

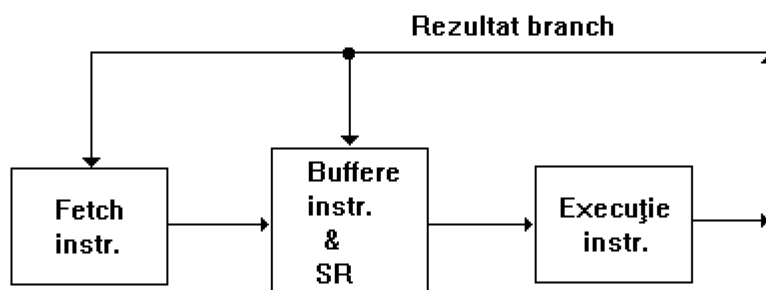


Figura 6.1. Arhitectura superscalara decuplata

Pe baze statistice se arata ca un basic-block contine, pe programele de uz general, doar 4-5 instructiuni în medie, ceea ce înseamna ca rata de fetch

a instructiunilor e limitata la cca. 5, aducerea simultana a mai multor instructiuni fiind inutila (fetch bottleneck). Desigur, aceasta limitare fundamentala ar avea consecinte defavorabile si asupra consumatorului, care ar limita principial si rata medie de executie a instructiunilor (IR - Issue Rate) la aceasta valoare. Progresele semnificative în algoritmii de lansare în executie impun însa depasirea acestei bariere. În acest sens, cercetarile actuale insista pe îmbunatatirea mecanismelor de aducere a instructiunilor prin urmatoarele tehnici:

- predictia simultana a mai multor ramificatii / tact rezultând deci rate IR sporite.
- posibilitatea accesarii si aducerii simultane a mai multor basic- block-uri din cache, chiar daca acestea sunt nealiniate, prin utilizarea unor cache-uri multiport
- pastrarea unei latente reduse a procesului de aducere a instructiunilor, în contradictie cu cele 2 cerinte anterioare.

Alti factori care determina limitarea ratei de fetch a instructiunilor (FR- Fetch Rate) sunt: largimea de banda limitata a interfetei procesor - cache, missurile în cache, predictiile eronate ale ramificatiilor, etc.

O paradigma interesanta, situata în prelungirea conceptului de superscalaritate si care poate constitui o solutie interesanta fata de limitarile mai sus mentionate, o constituie trace-procesorul, adica un procesor superscalar având o **memorie trace-cache** (TC). Ca si cache-urile de instructiuni (IC), TC este accesata cu adresa de început a noului bloc de instructiuni ce trebuie executat, în paralel cu IC. În caz de miss în TC, instructiunea va fi adusa din IC sau - în caz de miss si aici - din memoria principala. Spre deosebire însa de IC, TC memoreaza instructiuni contigue din punct de vedere al secventei lor de executie, în locatii contigue de memorie. O linie din TC memoreaza un segment de instructiuni executate dinamic si secvential în program (trace-segment). Evident, un trace poate contine mai multe basic-block-uri (unitati secventiale de program). Asadar, o linie TC poate contine N instructiuni sau M basic- block-uri, N>M, înscrise pe parcursul executiei lor.

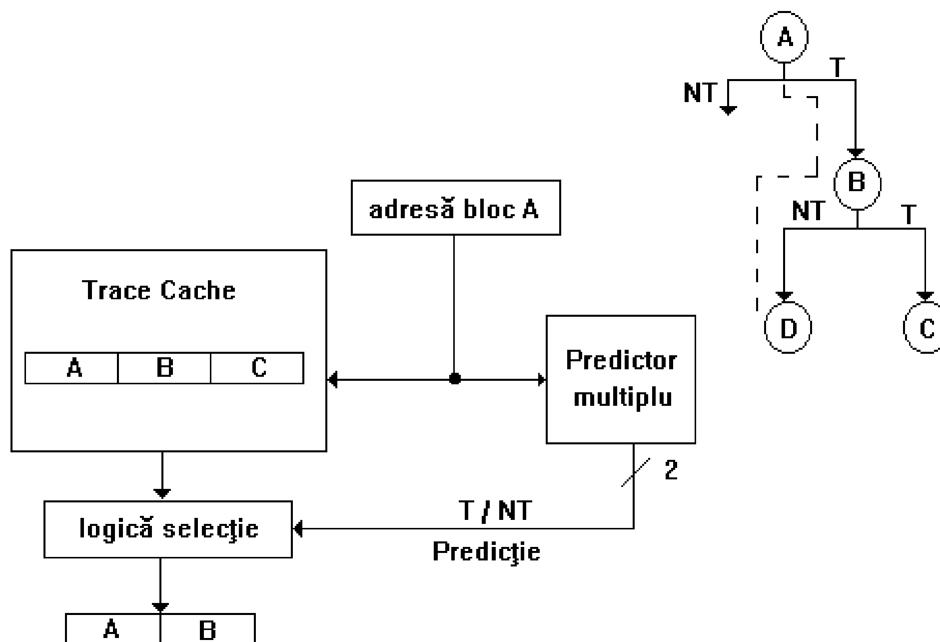


Figura 6.2. Ansamblul trace-cache respectiv predictor multiplu

Memoria TC este accesata cu adresa de început a basic-block-ului A, în paralel cu predictorul multiplu de salturi (vezi figura 6.2). Acesta, spre deosebire de un predictor simplu, predictioneaza nu doar adresa de început a urmatorului basic- block ce trebuie executat ci toate cele (M-1) adrese de început aferente urmatoarelor (M-1) basic- block-uri care urmeaza dupa A. Cei (M-1) biti generati de catre predictorul multiplu (taken/ not taken) selecteaza spre logica de executie doar acele blocuri din linia TC care sunt predictionate ca se vor executa (în cazul acesta doar blocurile A si B întrucât predictorul a selectat blocurile ABD ca se vor executa, în timp ce în linia TC erau memorate blocurile ABC).

O linie din TC contine [38]:

- N instructiuni în forma decodificata, fiecare având specificat blocul careia îi apartine.
- cele 2^{M-1} posibile adrese destinatie aferente celor M blocuri stocate în linia TC.
- un câmp care codifica numarul si "directiile" salturilor memorate în linia TC.

Înainte de a fi memorate în TC, instructiunile pot fi predecodificate în scopul înscrierii în TC a unor informatii legate de dependentele de date ce caracterizeaza instructiunile din linia TC curenta. Aceste informatii vor

facilita procese precum bypassing-ul datelor între unitatile de executie, redenumirea dinamica a registrilor cauzatori de dependente WAR (Write After Read) sau WAW (Write After Write) între instructiuni, etc., utile în vederea procesarii Out of Order a instructiunilor. O linie din TC poate avea diferite grade de asociativitate în sensul în care ea poate contine mai multe pattern-uri de blocuri, toate având desigur aceeasi adresa de început (A), ca în figura 6.3.

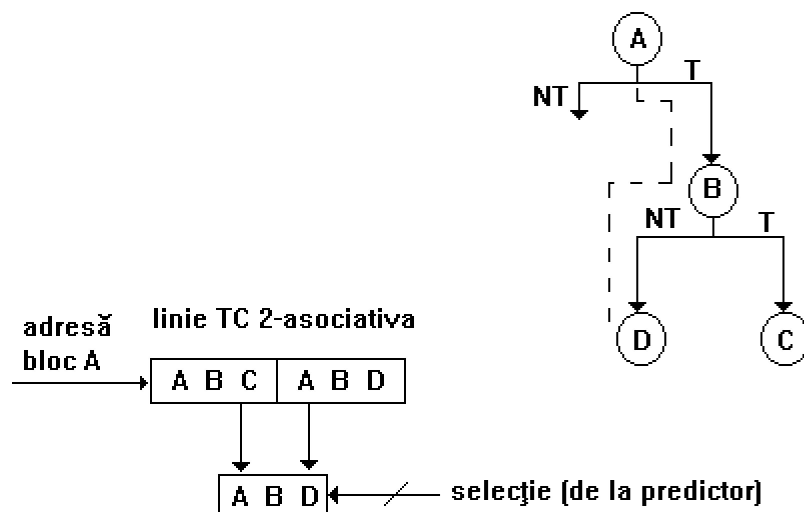


Figura 6.3. Selectia dintr-o linie trace-cache asociativa

Asadar, segmentele începând de la aceeasi adresa (A), sunt memorate în aceeasi linie asociativa din TC. Ca si în structurile TC neasociative, verificarea validitatii liniei selectate se face prin compararea (cautarea) dupa tag. Deosebirea de esenta consta în faptul ca aici este necesara selectarea - în conformitate cu pattern-ul generat de catre predictorul multiplu - trace-ului cel mai lung dintre cele continute în linia respectiva. Este posibil ca aceasta selectie complexa sa dureze mai mult decât în cazul neasociativ si prin urmare sa se repercuteze negativ asupra duratei procesului de aducere a instructiunilor (fetch). Avantajul principal însa, dupa cum se observa si în figura, consta în faptul ca este probabil sa se furnizeze procesorului un numar de blocuri "mai lung" decât un TC simplu. Astfel de exemplu, daca pattern-ul real de blocuri executate este ABD, structura TC îl va furniza fara probleme, în schimb o structura TC neasociativa ce contine doar pattern-ul ABC, evident va furniza în aceasta situatie doar blocurile AB.

Pe masura ce un grup de instructiuni este procesat, el este încarcat într-o asa-numita "fill unit" (FU-unitate de pregatire). Rolul FU este de a asambla instructiunile dinamice, pe masura ce acestea sunt executate, într-un trace-segment. Segmentele astfel obtinute sunt memorate în TC. Dupa cum am mai subliniat, este posibil ca înainte de scrierea segmentului în TC, FU sa analizeze instructiunile din cadrul unui segment spre a marca explicit dependentele dintre ele. Acest lucru va usura mai apoi lansarea în executie a acestor instructiuni întrucât ele vor fi aduse din TC si introduse direct în statiile de rezervare aferente unitatilor functionale. Unitatea FU se ocupa deci de colectarea instructiunilor lansate în executie, asamblarea lor într-un grup de N instructiuni (sau M blocuri) si înscrierea unui asemenea grup într-o anumita linie din TC. Exista desigur cazuri când FU poate crea copii multiple ale unor blocuri în TC. Aceasta redundanta informationala poate implica degradari ale performantei, dar pe de alta parte, lipsa redundantei ar degrada valoarea ratei de fetch a instructiunilor deci si performanta globala.

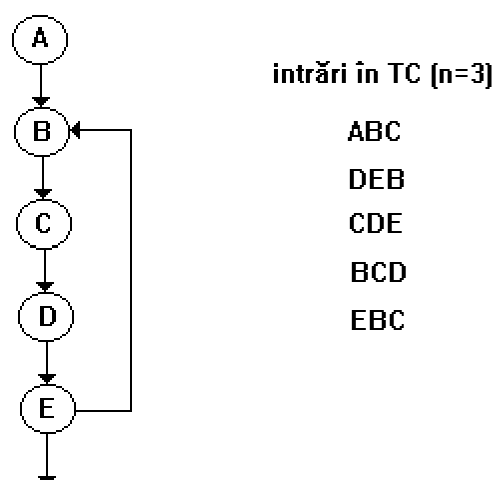


Figura 6.4. Segmente asamblate pe timpul executiei unei bucle de program

Se poate deci afirma ca un TC exploateaza reutilizarea eficienta a secventelor dinamice de instructiuni, reprocesate frecvent în baza a 2 motive de principiu: localizarea temporala a trace-ului si respectiv comportarea predictibila a salturilor în virtutea comportarii lor anterioare. Asadar, TC memoreaza trace-uri în scopul eficientizarii executiei programului si nu doar în scopul eficientizarii procesului de aducere al instructiunilor. Aceasta, pe motiv ca un segment din trace contine numai instructiuni care se vor executa. În cazul IC, daca într-un bloc exista o ramificatie efectiva, instructiunile urmatoare se aduceau inutil întrucât nu s-ar fi executat.

Cum TC trebuie sa lucreze într-o strânsa dependenta cu predictorul de salturi, se impune îmbunatatirea performantelor acestor predictoare. Se pare ca solutia de viitor va consta într-un predictor multiplu de salturi, al carui rol principal consta în predictia simultana a urmatoarelor (M-1) salturi asociate celor maximum M blocuri stocabile în linia TC. De exemplu, pentru a predictiona simultan 3 salturi printr-o schema de predictie corelata pe 2 nivele, trebuie expandata fiecare intrare din structura de predictie PHT (Pattern History Table), de la un singur numarator saturat pe 2 biti, la 7 astfel de automate de predictie, ca în figura 6.5. Astfel, predictia generata de catre primul predictor (taken / not taken) va multiplexa rezultatele celor 2 predictoare asociate celui de al doilea salt posibil a fi stocat în linia curenta din TC. Ambele predictii aferente primelor 2 salturi vor selecta la rândul lor unul dintre cele 4 predictoare posibile pentru cel de-al treilea salt ce ar putea fi rezident în linia TC, predictionându-se astfel simultan mai multe salturi. Daca predictorul multiplu furnizeaza simultan mai multe PC-uri, TC rezolva elegant si problema aducerii simultane a instructiunilor pointate de aceste PC-uri, fara multiportarea pe care un cache conventional ar fi implicat-o.

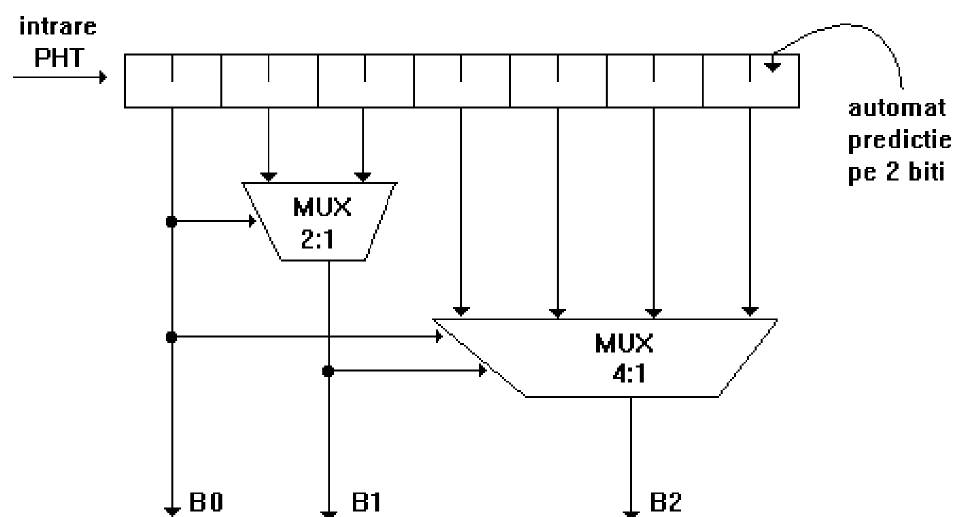


Figura 6.5. Predictor a 3 salturi succesive

Asemenea predictoare multiple în conjunctie cu structuri de tip TC conduc practic la o noua paradigma a procesarii unui program masina numita "multiflow", caracterizata deci prin procesarea în paralel a mai multor basic-block-uri dintr-un program. În [38] se prezinta o cercetare bazata pe simulare asupra conceptelor novatoare de TC si predictor

multiplu, integrate într-o arhitectura superscalara extrem de agresiva dezvoltata la Universitatea din Michigan, SUA. În esenta, investigatia subliniaza urmatoarele aspecte:

- ♦ cresterea gradului de asociativitate a TC de la 0 (mapare directa) la 4 (asociativitate în blocuri de 4 intrari/ bloc) poate duce la cresteri ale ratei medii de procesare a instructiunilor de pâna la 15%
- ♦ capacitati egale ale TC si respectiv memoriei cache de instructiuni (64 ko, 128 ko) conduc la performante cvasioptimale
- ♦ asociativitatea liniei TC nu pare a conduce la cresteri spectaculoase de performanta
- ♦ performanta globala fata de o arhitectura echivalenta, dar fara TC, creste cu circa 24%, iar rata de fetch a instructiunilor a instructiunilor în medie cu 92%.

La ora actuala, procesorul Intel Pentium IV reprezinta primul procesor comercial care înlocuieste nivelul L1 de cache clasic cu un *Execution Trace Cache*. De asemenea, alte caracteristici arhitecturale pentru respectivul procesor constituie: integrarea a 42 milioane de tranzistori, un pipeline ce poate functiona pe 20 de nivele, expedierea simultana spre executie a 4 instructiuni per perioada de tact procesor, o magistrala ce va functiona la frecventa de 400 MHz, rata de transfer la memorie ajungând astfel la 3,2 Gb/s [54].

În [39] se prezinta o alta tehnica de procesare, legata tot de **reutilizarea dinamica a instructiunilor** deja executate, posibil a fi folosita în conjunctie cu un trace cache. Aici însa, principalul scop urmarit consta în paralelizarea executiei unor instructiuni dependente RAW, bazat pe predictia valorilor registrilor utilizati de aceste instructiuni. Ideea originara apartine scolii de arhitectura calculatoarelor de la Universitatea din Wisconsin – Madison, mai precis cercetatorilor *A. Sodani si G. Sohi* care au introdus în 1997, la conferinta ISCA 97 tinuta la Denver, SUA, conceptul de reutilizare dinamica a instructiunilor – bazat pe o noua tehnica microarhitecturala, *non-speculativa* menita sa exploateze fenomenul de repetitie dinamica a instructiunii, reducând cantitatea de cod necesar a fi executat. Autorii arata în primul rând ca reutilizarea unor instructiuni sau secvente de instructiuni este relativ frecventa si se datoreaza modului compact de scriere al programelor precum si caracteristicilor intrinseci ale structurilor de date prelucrate. O instructiune dinamica este repetata daca ea opereaza asupra acelorasi intrari si produce aceleasi rezultate (iesiri) precum o instanta anterioara a aceleiasi instructiuni. Ideea de baza este ca daca o secventa de instructiuni se reia în acelasi “context de intrare”, atunci executia sa nu mai are sens fiind suficienta o simpla actualizare a

"contextului de iesire", în concordanta cu cel precedent. Se reduce astfel numarul de instructiuni executate dinamic prin asigurarea unui suport arhitectural care transforma executia instructiunilor în cautari (citiri) în tabele hardware. Daca arhitectura TC - dupa cum am aratat - actioneaza în principal asupra depasirii unei limitari fundamentale asupra ratei de fetch a instructiunilor, aceasta noua inovatie arhitecturala va actiona asupra depasirii limitarii ratei de executie a instructiunilor. Reamintim ca aceasta rata de executie este fundamental limitata de hazardurile structurale implicate precum si de hazardurile RAW între instructiuni. Asadar, instructiunile reutilizate nu se vor mai executa din nou, ci pur si simplu contextul procesorului va fi actualizat în conformitate cu actiunea acestor instructiuni, bazat pe istoria lor memorata.

În [41] se analizeaza mai întâi daca gradul de reutilizare (si repetabilitate) al instructiunilor dinamice este semnificativ si se arata ca raspunsul este unul afirmativ. Mai putin de 20% din numarul instructiunilor statice care sunt repetate (genereaza cel putin o instructiune dinamica repetata) implica o repetabilitate de peste 90% a instructiunilor dinamice. Exista în acest sens 2 cauze calitative: în primul rând faptul ca programele sunt scrise în mod *generic*, ele operând asupra unei varietati de date de intrare, iar în al doilea rând, aceste programe sunt scrise într-un mod *concis* – mentinerea unei reprezentari statice compacte a unei secvente dinamice de operatii – în vederea obtinerii rezultatelor dorite (aici structurile de tip "*bucla*" sunt reprezentative).

Pentru o mai buna întelegere a fenomenului de repetitie a instructiunilor, executia dinamica a programelor este analizata la trei nivele: *global*, de *functie* si *local* (în interiorul functiei). În analiza globala, paternurile de date utilizate în programe sunt retinute ca entitati întregi si determinate sursele de repetitie a instructiunilor (intrari externe, initializari globale de date sau valori interne ale programelor). Întrucât repetitia instructiunilor, conform rezultatelor obtinute de autori, se datoreaza în mare masura ultimelor doua surse de repetitie (valori imediate si initializari de date globale) se impune concluzia ca fenomenul de repetitie este mai mult o proprietate a modului în care calculul (operatiile efectuate) este exprimat în program si mai putin o proprietate a datelor de intrare. Concluziile generate în urma analizei la nivel de functie sunt ca de foarte multe ori functiile sunt invocate repetat cu exact aceleasi valori ale parametrilor de intrare si ca relativ putine apeluri de functii nu au argumente repetate. Chiar si în cazul unor apeluri repetate ale unei functii cu parametrii de intrare diferiti, procentajul de instructiuni dinamice reutilizabile poate fi semnificativ. La nivelul analizei locale, instructiunile functiilor/procedurilor sunt clasificate în functie de sursa valorilor datelor folosite (ex: argumentele functiei, date

globale, valori returnate de alte functii) si functie de sarcina realizata (ex: salvare restaurare registri, prolog, epilog, calcul adrese globale). Majoritatea repetitiei instructiunilor se datoreaza valorilor globale sau argumentelor functiei dar si functiilor prolog si epilog.

Preluat din [40], se prezinta în continuare un exemplu sugestiv în care apare fenomenul de reutilizare dinamica a instructiunilor.

Functia *func* (figura 6.6.a) cauta o valoare *x* în tabloul *list* de dimensiunea *size*. Functia principala *main_func* (figura 6.6.c) apeleaza functia *func* de mai multe ori, cautând câte un alt element în acelasi tablou la fiecare apel. La apelul functiei *func* tabloul este parcurs element cu element în mod iterativ, cautându-se valoarea pâna la capatul tabloului, conditia de încheiere a cautarii reprezentând-o gasirea elementului. Expandarea buclei din interiorul functiei *func* corespondenta unei iteratii este prezentata în figura 6.6.b. Instantele dinamice ale instructiunilor generate de primul apel *func* sunt descrise în figura 6.6.d. În fiecare iteratie a buclei, instructiunea 2 este dependenta de parametrul *size*, instructiunile 3 si 4 sunt dependente de parametrul *list*, instructiunea 5 este dependenta atât de *list* cât si de valoarea cautata în tablou, iar instructiunea 6 este dependenta de contorul i. Daca *func* e apelata din nou în acelasi tablou *list* (de aceeasi dimensiune *size*), dar cu alt parametru de cautare, atunci toate instantele dinamice ale instructiunilor 1÷4 si 6 vor produce aceleasi rezultate pe care le-au produs la apelul anterior al functiei *func*. Doar instantele dinamice ale instructiunii 5 produc rezultate care ar putea diferi de apelurile anterioare ale functiei *func*. Repetarea rezultatelor instantelor dinamice ale instructiunilor 1÷4 si 6 este direct atribuita faptului ca *func* a fost scrisa ca o functie generica de cautare într-un tablou, dar în acest caz particular, doar unul din parametri se modifica între apeluri diferite. Chiar daca *func* ar fi apelata cu toti parametri diferiti pentru fiecare apel în parte, instantele dinamice diferite ale instructiunii 6 (i=0, i=1, i=2,…) vor produce aceleasi valori generate în primul apel al functiei, consecinta a utilizarii buclelor pentru a exprima calculele dorite într-o maniera concisa. Totusi, daca parametrul *size* ar fi diferit la un al doilea apel al functiei *func*, atunci doar *min*(*size1*, *size2*) instante dinamice ale instructiunii 6 vor produce aceleasi rezultate. Prin urmare, acest exemplu sugestiv arata faptul ca repetabilitatea instructiunilor dinamice este considerabila si în consecinta reutilizarea instructiunilor este posibila.

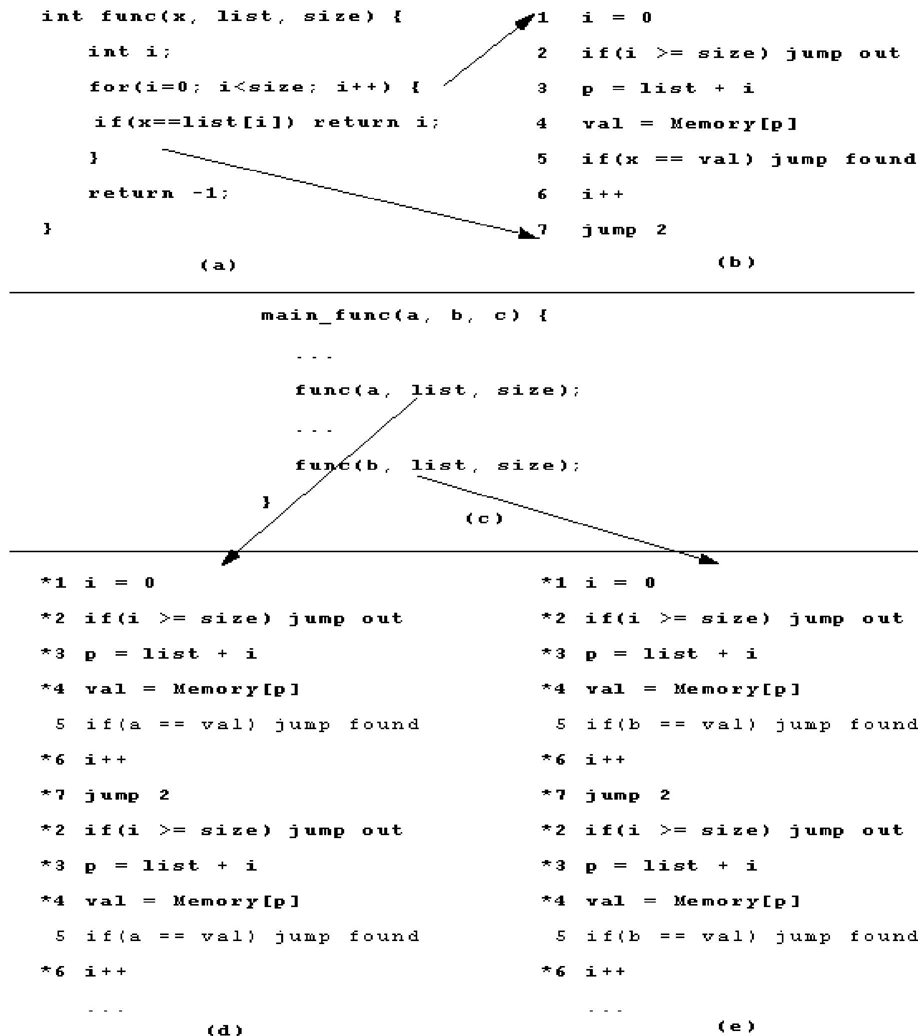


Figura 6.6. Exemplu ilustrând repetabilitatea instructiunilor

Instantele dinamice marcate "*" vor realiza aceleasi operatii pentru ambele apeluri ale functiei *func*.

Bazat pe aceste premise autorii [40] dezvolta 3 scheme de reutilizare dinamica a instructiunilor, primele 2 la nivel de instructiune iar ultima, la nivel de lant de instructiuni dependente. Instructiunile deja executate, se memoreaza într-un mic cache numit buffer de reutilizare (Reuse Buffer - RB). Acesta poate fi adresat (cautat) cu PC-ul având si un mecanism pentru invalidarea selectiva a unor intrari bazat pe actiunile anumitor evenimente. Desigur ca RB trebuie sa permita si un mecanism de testare a reutilizabilitatii instructiunii selectate.

RB poate avea o structura asociativa; cu cât gradul de asociativitate este mai mare cu atât numarul instantelor dinamice ale unei instructiuni care sunt memorate în RB la un moment dat este mai mare. În acest caz, cautarea în RB trebuie facuta dupa o informatie de context mai bogata care sa permita identificarea unei instante dinamice a instructiunii statice adresate curent. Un exemplu de astfel de informatie ar putea consta în valorile registrilor sursa respectiv destinatie etc.

Testul de reutilizare verifica daca informatia accesata din RB reprezinta un rezultat reutilizabil. Detaliile de implementare ale testului depind de fiecare schema de reutilizare folosita. De asemenea, trebuie tratate doua aspecte privind managementul RB: stabilirea instructiunii care va fi plasata în buffer si mentinerea consistentei bufferului de reutilizare. Decizia privind modul de inserare a instructiunilor în RB poate varia de la una nerestrictiva ("*no policy*"), care plaseaza toate instructiunile în buffer, în cazul în care nu sunt prezente deja, la una mai selectiva care filtreaza instructiunile ce vor fi inserate dupa probabilitatea statistica de a fi reutilizate. Problema consistentei are în vedere garantarea corectitudinii rezultatului instructiunii reutilizate din RB. Mentinerea consistentei informatiilor în RB depinde de fiecare schema de reutilizare în parte dupa cum se va putea constata în cele ce urmeaza.

În vederea compatibilizarii cu modelul superscalar care lanseaza în executie mai multe instructiuni simultan, RB este în general multiport pentru a putea permite reutilizarea mai multor instructiuni de executie curenta. Este evident ca, gradul de multiportare al RB-ului nu are sens a fi mai mare decât fereastra maxima de executie a instructiunilor.

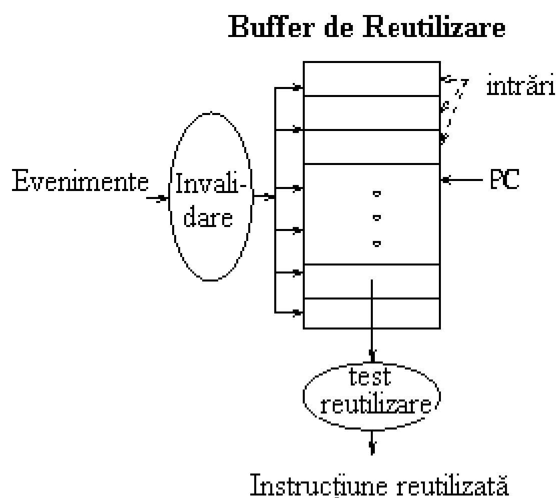


Figura 6.7. Structura hardware a bufferului de reutilizare

Este indexat cu PC-ul instructiunii. Prezinta un mecanism de invalidare selectiva a intrarilor pe baza unor evenimente.

În cazul reutilizarii la nivel de instructiune, o intrare în RB ar putea avea urmatorul format:

Tag	Op1	Op2	Adr	Rez	Rez_valid	Mem_Valid

Tag – ar putea fi reprezentat în esenta de catre PC-ul instructiunii.

Op1, **Op2** – reprezinta numele registrilor utilizati de catre instructiune.

Rez – reprezinta rezultatul actual al instructiunii, cel care va fi reutilizat în caz de "*hit*" în bufferul RB.

Rez_Valid – indica daca rezultatul "*Rez*" este valid sau nu. Este setat odata cu introducerea instructiunii în RB. Este resetat de catre orice instructiune care scrie într-unul din registrii sursa (Op1, Op2).

Adr – este adresa (reutilizabila) de memorie în cazul unei instructiuni Load/Store.

Mem_Valid – indica daca valoarea din câmpul "*Rez*" este reutilizabila în cazul unei instructiuni Load. Bitul este setat la înscrierea instructiunii Load în RB. Resetarea bitului se face prin orice instructiune Store având aceeasi adresa de acces.

Rezulta ca pentru instructiunile aritmetico-logice reutilizarea este asigurata daca *Rez_Valid*=1. De asemenea, *Rez_Valid*=1 garanteaza adresa corecta pentru orice instructiune Load/Store si deci scuteste procesorul de calculul ei. În schimb rezultatul unei instructiuni Load nu poate fi reutilizat decât daca *Mem_Valid*=1 si *Rez_Valid*=1. Plusul de performanta datorat reutilizarii dinamice a instructiunilor se datoreaza atât scurtcircuitarii unor nivele din structura "pipe" cât si reducerii hazardurilor structurale si deci a presiunii asupra diverselor resurse hardware. Astfel, prin reutilizarea instructiunilor se evita stagnarea în statiile de rezervare (Instruction Window) si timpul de executie, rezultatele instructiunilor reutilizate fiind scrise în bufferul de reordonare. Rezulta de asemenea, o disponibilizare a unitatilor functionale de executie care nu vor mai avea de procesat instructiunile reutilizate si eventual o deblocare a instructiunilor dependente RAW de cea reutilizata.

O structura de RB mai complexa prezentata în [40], care memoreaza si dependentele de date RAW între diferitele instructiuni, permite reutilizarea la nivel de lant de instructiuni dependente. Cresterea de performanta implicata de astfel de scheme este de cca. 7,5% ÷ 15%. În schimb procentajul instructiunilor reutilizate are valori medii cuprinse între 2% ÷ 26%.

În vederea reutilizarii unor lanturi de instructiuni dependente RAW, pe parcursul procesarii instructiunilor, se construiesc în mod dinamic asa-numite seturi de instructiuni. O instructiune "i" se adauga unui set notat cu S daca "i" depinde RAW de cel putin una dintre instructiunile setului S. În caz contrar, instructiunea "i" va fi prima apartinând unui nou set. Practic, constructia acestor seturi implica generarea grafului dependentelor de date atasat programului, ca în secventa de mai jos preluata din [39] (vezi figura 6.8).

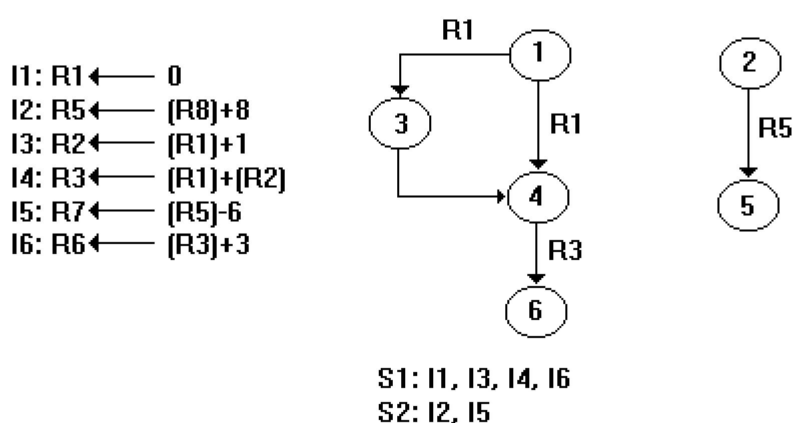


Figura 6.8. Constructia seturilor în vederea reutilizarii codurilor

Dupa procesarea instructiunilor, seturile rezultate sunt înscrise în vederea reutilizarii într-un buffer special numit TDIS (Table of Dependent Instruction Sequences). O intrare în TDIS contine 3 parti principale:

- partea IN, care memoreaza valorile operanzilor de intrare, adica aceia neprodusi prin secventa respectiva ci preluati din afara acesteia.
- partea INSTRUCTION, contine adresele instructiunilor inserate în seturi.
- partea OUT, ce contine numele registrilor destinatie aferenti unui set, precum si valorile acestora.

Pentru exemplificare, secventa de program anterioara necesita un buffer TDIS cu doua intrari, ca mai jos (figura 6.9).

IN		INSTRUCTION				OUT				
		I1	I3	I4	I6	R1=0		R2=1	R3=1	R6=4
R8	9	I2	I5			R5=17	R7=11			

Figura 6.9. Structura TDIS la un moment dat

Asadar, la fiecare aducere a unei noi instructiuni, PC-ul acesteia se compara cu adresa primei instructiuni din fiecare linie a TDIS. Apoi, continutul actual al registrilor procesorului este comparat cu cel al partii IN a TDIS. În caz de hit, secventa de instructiuni din TDIS poate fi reutilizata cu succes si cu eludarea tuturor hazardurilor RAW dintre aceste instructiuni. Executia acestor instructiuni va însemna doar actualizarea contextului procesorului în conformitate cu valorile OUT din TDIS. Prin urmare, reutilizarea instructiunilor prin acest mecanism va avea un efect benefic asupra timpului de procesare al instructiunilor. Considerând un procesor superscalar care poate aduce, decodifica si executa maximum 4 instructiuni / ciclu, secventa anterioara se proceseaza ca în cele doua figuri urmatoare (6.10**,** 6.11).

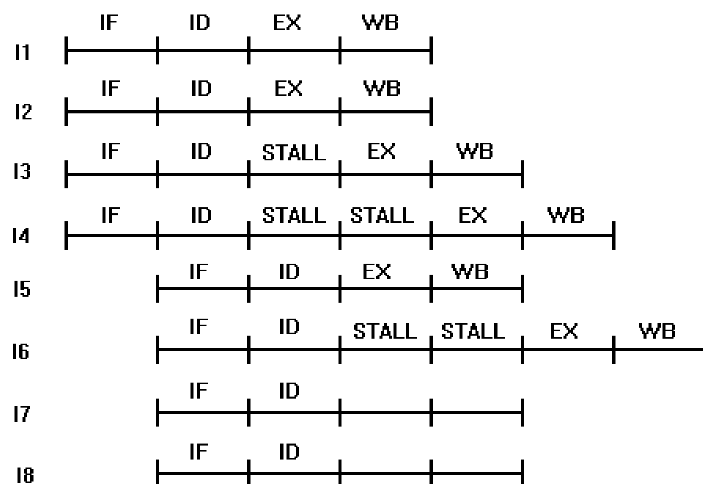


Figura 6.10. Executia programului pe un procesor superscalar

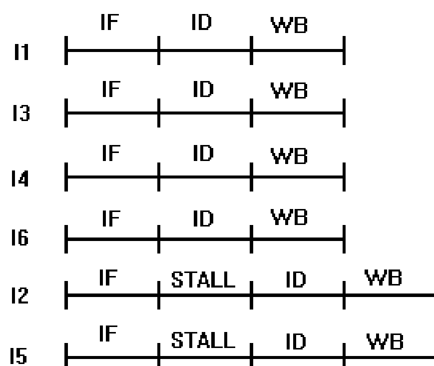


Figura 6.11. Executia programului pe un procesor cu reutilizarea codurilor

Se observa ca bufferul TDIS determina executia secventei prin reutilizarea instructiunilor în doar 4 cicli fata de 7 cicli câti ar fi fost necesari în cazul unei procesari clasice. Daca largimea de banda a decodorului de instructiuni ar fi fost de 6 instructiuni în loc de 4, executia secventei s-ar fi redus la doar 3 cicli. Teste efectuate pe benchmark-urile SPEC 95 si prezentate în [39], au aratat ca între 17% si 26% dintre instructiunile acestor programe au putut fi reluate cu succes. Conceptul TDIS este eficient întrucât ca si în cazul utilizarii instructiunilor combinate [42], se elimina necesitatea secventierii în executie a unor instructiuni dependente RAW. Mai mult, în opinia autorilor, dintr-un anume punct de vedere, conceptul reutilizarii dinamice a secventelor dependente de instructiuni, violeaza oarecum celebra lege a lui *G. Amdahl* întrucât trece peste secventialitatea intrinseca a programului si proceseaza agresiv paralel chiar si în acest caz, prin “updating”. Este fara îndoiala posibil ca acest concept sa se cupleze favorabil cu cel de tip "trace cache" anterior prezentat si care actioneaza favorabil în special asupra limitarilor ratei de fetch a instructiunilor.

Figura 6.11 ilustreaza o microarhitectura tipica cu reutilizarea instructiunilor. Singura modificare de principiu fata de modelul superscalar este data de aparitia bufferului de reutilizare. În faza *fetch instructiune* sunt extrase din cache-ul de instructiuni sau memoria principala instructiuni si plasate în bufferul de prefetch (Instruction Queue). Urmeaza apoi faza de *decodificare* a instructiunilor *si redenumire* a registrilor. În faza *citire operand* valorile operanzilor aferenti instructiunilor sunt citite fie din setul de registri generali fie din bufferul de reordonare, functie de structura care contine ultima versiune a registrilor. Accesul la Bufferul de Reutilizare poate fi pipelineizat si suprapus cu faza fetch instructiune. Imediat dupa decodificarea instructiunii, în timpul fazei de citire operanzi se realizeaza

testul de reutilizare asupra intrarilor citite din RB pentru a sti daca rezultatele instructiunilor sunt reutilizabile. Daca este gasit un rezultat reutilizabil instructiunea aferenta nu mai trebuie procesata în continuare si se evita rutarea acestora în *fereastra de instructiuni trimise spre procesare* (Instruction Window) si este transmis direct bufferului de reordonare. Instructiunile Load evita fereastra Instruction Window doar daca rezultatele ambelor micro-operatii (calculul adresei si accesarea memoriei) sunt reutilizate. Testarea reutilizarii poate dura unul sau mai multi cicli. În cel din urma caz, instructiunile sunt plasate în Issue Window, pe durata realizarii testului de reutilizare. Daca instructiunea nu s-a executat înca în momentul încheierii testului de reutilizare (din cauza ca nu au existat unitati functionale de executie aferente disponibile), rezultatul obtinut din RB este folosit iar instructiunea este retrasa din procesare spre bufferul de reordonare. Daca totusi instructiunea s-a executat complet înainte de încheierea testului de reutilizare, atunci rezultatul reutilizabil este ignorat. În ambele situatii, daca nu se gaseste un rezultat reutilizabil în RB, atunci se aloca intrarea corespunzatoare din RB unde rezultatul instructiunii va fi plasat dupa executie – asadar o **inserare non-speculativa** -, în vederea unei eventuale reutilizari ulterioare mentinând consistenta si a altor structuri hardware daca e necesar (tabela RST din schema de reutilizare cu nume si dependente: S_{n+d}).

În cazul unei predictii eronate a unei instructiuni de ramificatie, mecanismul de refacere a contextului va trebui sa fie suficient de selectiv astfel încât sa nu invalideze în RB instructiunile situate imediat dupa punctul de convergenta al ramificatiei si care ar putea fi reutilizate. Astfel, reutilizarea este exploatata la maxim si în acest caz, cu beneficii evidente asupra performantei.

Rezumând, se desprind câteva avantaje introduse de tehnica de reutilizare dinamica a instructiunilor:

- Scurtcircuitarea unor nivele din structura pipe de catre instructiunile reutilizate, reducând presiunea asupra resurselor (statii de rezervare, unitati functionale, porturi ale cache-urilor de date) necesare altor instructiuni aflate în asteptare.
- La reutilizarea unei instructiuni rezultatul sau devine cunoscut mai devreme decât în situatia în care s-ar procesa normal, permitând în consecinta altor instructiuni dependente de aceste rezultate sa fie executate mai rapid.
- Reduce penalitatea datorata predictiei eronate a adreselor destinatie în cazul instructiunile de salt, prin reutilizarea codului succesor punctului de convergenta.

- Colapsarea dependentelor de date determina îmbunatatirea timpului de executie al instructiunilor crescând gradul de paralelism al arhitecturii.
- Procentajul de reutilizare al instructiunilor dinamice calculat pe benchmark-urile SPEC ’95 [40] este semnificativ, ajungându-se la valori maxime de 76%.
- Speed-up-ul obtinut fata de modelul superscalar [40] pe aceleasi programe de test nu este la fel de pronuntat ca procentul de reutilizare, valoarea maxima atinsa fiind de 43%.

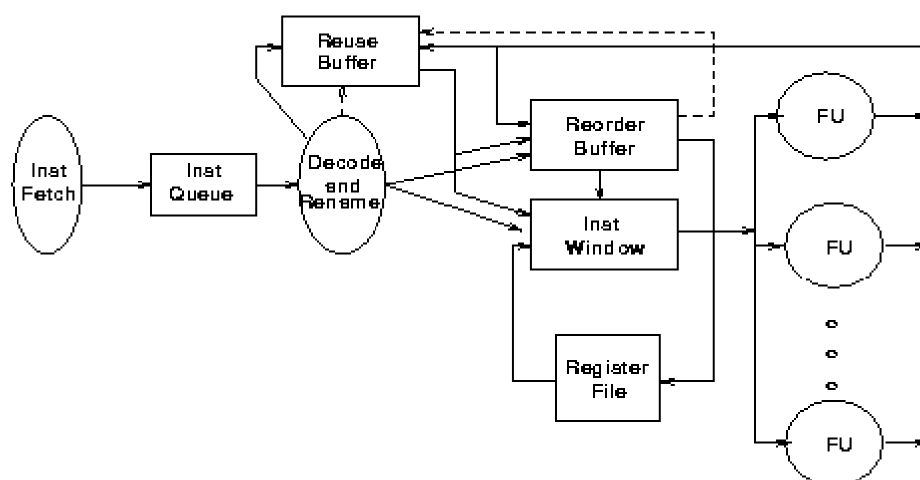


Figura 6.12. Microarhitectura generica cu buffer de reutilizare

O alta tehnica hardware, si ea relativ recent dezvoltata, care în mod similar cu *reutilizarea dinamica a instructiunilor* urmareste exploatarea redundantei existente în programe prin colapsarea dinamica a dependentelor de date o reprezinta *predictia valorilor* (Value Prediction). Desi ambele tehnici urmaresc reducerea timpului de executie al programelor reducând sau eliminând în totalitate constrângerile legate de fluxul de date (*dataflow*), exista totusi diferente (chiar majore privind unele aspecte) legate de modul de interactiune al fiecarei tehnici în parte cu celelalte caracteristici microarhitecturale. De asemenea, se pun probleme relativ la modul de determinare **speculativa** (predictia valorii) sau **non-speculativa** (reutilizarea instructiunilor) a redundantei în programele de uz general, avantajele si dezavantajele implicate de fiecare tehnica, cantitatea de redundanta captata de fiecare tehnica în parte etc. Înainte de a face cunoscute si a întelege diferentele existente între predictia valorilor si reutilizarea instructiunilor vom descrie câteva caracteristici si aspecte legate de conceptul de *localitate a valorii* si predictia valorilor.

Localitatea valorii reprezinta o a treia dimensiune a conceptului de *localitate* (pe lânga cea *temporala* si respectiv *spatiala*), frecvent întâlnita în programele de uz general si utilizata în predictia valorilor descriind probabilitatea statistica de referire a unei valori anterior folosite si stocata în aceeasi locatie de memorie. Conceptul de localitate a valorii – introdus de M. Lipasti [43] – este strâns legat de *calculul redundant* (repetarea executiei unei operatii cu aceeasi operanzi) [44] si este exploatat de o serie de tehnici de predictia valorilor dintre care amintim *Load Value Prediction* (LVP) [43] si *Branch Prediction* (într-un sens mai restrictiv) [45]. Convingerea ca "*localitatea valorilor*" exista are la baza rezultate statistice obtinute prin simulare la nivel de executie a instructiunilor pe benchmark-uri SPEC '95 [43]. Cu o "adâncime" a istoriei de predictie de 1 (regasirea aceleiasi valori în resursa asignata ca si în cazul precedentului acces), programele de test exprima o localitate a valorii de 50% în timp ce extinzând verificarea în spatiul ultimelor 16 accese la memorie se obtine o localitate de 80%. Rezultatele subliniaza ca majoritatea instructiunilor Load statice aferente unui program exprima o variatie redusa a valorilor pe care le încarca pe parcursul executiei.

Tehnica LVP predictioneaza rezultatele instructiunilor Load la expedierea spre unitatile functionale de executie exploatând corelatia dintre adresele respectivelor instructiuni si valorile citite din memorie de catre acestea, permitând deci instructiunilor Load sa se execute înainte de calculul adresei si îmbunatatind astfel performanta. Conceptul de localizare a valorilor se refera practic la o corelatie dinamica între numele unei resurse (registru, locatie de memorie, port I/O) si valoarea stocata în acea resursa. Daca memoriile cache conventionale se bazeaza pe localitatea temporala si spatiala a datelor pentru a reduce timpul mediu de acces la memorie, tehnica LVP exploateaza localitatea valorii reducând atât timpul mediu de acces la memorie cât si necesarul de largime banda al memoriei (se efectueaza mai putine accese la memoria centrala) asigurând un câstig de performanta considerabil. Desigur ca, toate aceste avantaje se obtin simultan cu reducerea considerabila a presiunii asupra memoriilor cache. Ca si consecinta a predictiei valorilor se reduc si efectele defavorabile ale hazardurilor RAW, prin reducerea asteptarilor instructiunilor dependente ulterioare.

Localitatea valorilor este justificata de câteva observatii empirice desprinse din programele de uz general, medii si sisteme de operare diverse:

- *Redundanta datelor* – seturile de intrari de date pentru programele de uz general sufera mici modificari (Ex: matrici rare, fisiere text cu spatii goale, celule libere în foi de calcul tabelar).

- *Verificarea erorilor* – tehnica LVP poate fi benefica în gestionarea tabelelor de erori ale compilatoarelor, în cazul aparitiei unor erori repetate.
- *Constante în program* – deseori este mult mai eficient ca programele (codul) sa încarce constante situate în structuri de date din memorie, ceea ce este exploatat favorabil de tehnica LVP.
- *Calculului adreselor instructiunilor de salt* – în situatia instructiunilor *case* (*switch* în C) compilatorul trebuie sa genereze cod care încarca într-un registru adresa de baza pentru branch, care este o constanta (predictia adreselor destinatie pentru instructiunile de salt).
- *Apelul functiilor virtuale* – în acest caz compilatorul trebuie sa genereze cod care încarca un pointer de functie, care este o constanta în momentul rularii.

Localitatea valorii aferenta unor instructiuni Load statice dintr-un program poate fi afectata semnificativ de optimizarile compilatoarelor: *loop unrolling*, *software pipelining*, *tail replication*, întrucât aceste optimizari creeaza instante multiple ale instructiunilor Load. Ca si metrica de evaluare, localitatea valorii pentru un benchmark este calculata ca raport dintre numarul de instructiuni Load *statice* care regasesc o aceeasi valoare în memorie ca si precedentele k accese si numarul de instructiuni Load *dinamice* existente în benchmarkul respectiv. O istorie de localizare pe k biti semnifica faptul ca o instructiune Load verifica daca valoarea citita (încarcata) din memorie se regaseste printre ultimele k valori anterior încarcate. O problema importanta de proiectare care se pune la ora actuala este: *cât de "multa istorie" sa fie folosita în predictie (executie speculativa)* ? Compromisurile actuale care se fac sunt – o istorie redusa – reprezentând o acuratete de predictie joasa dar cost scazut sau – o istorie bogata de predictie – acuratete ridicata de predictie dar costuri hardware si regie ridicata.

Se remarca aici o similitudine între problema predictiei valorilor si problema - actuala si ea - a predictiei adreselor destinatie aferente instructiunilor de salt indirect. Structurile de date implementate în hardware pentru ambele procese de predictie, au acelasi principiu de functionare si anume: asocierea cvasibijectiva a contextului de aparitie al instructiunii respective (Load sau Branch) cu data / adresa de predictionat, în mod dinamic, odata cu procesarea programului. Iata deci ca problematica predictiei în microprocesoarele avansate, tinde sa devina una generala si ca urmare implementata pe baza unor principii teoretice mai generale si mai elevate. Aceasta are drept scop principal si imediat, executia speculativa

agresiva a instructiunilor, cu beneficii evidente în cresterea gradului mediu de paralelism.

Unul din mecanismele hardware de exploatare a conceptului de localitate a valorilor îl reprezinta LVPU (Load Value Prediction Unit) – unitate de predictie a valorilor încarcate prin instructiuni Load si abordeaza doua aspecte: latenta memoriei si respectiv largimea de banda a memoriei. Exploatând corelatia între adresele instructiunilor Load si valorile ce vor fi (sau sunt) încarcate (citite) de la respectiva adresa, se va reduce latenta instructiunilor Load. Prin identificarea celor mai predictibile Load-uri si bypassing-ul aplicat astfel ierarhiei de memorie conventionale: CPU – cache – memorie centrala, se reduce necesarul de largime de banda al memoriei (numarul de porturi de acces la memorie). Intentia conceptului de localitate a valorii este de a reduce efectul defavorabil al dependentelor de date RAW prin reducerea latentei memoriei la zero cicli (nefiind necesar accesul la memoria principala) precum si reducerea timpilor de asteptare aferenti instructiunilor dependente urmatoare.

Componentele LVPU sunt:

- ⇨ **LVPT** (Load value prediction table) – un tabel utilizat în generarea valorilor de predictie. Câmpurile în acest tabel sunt constituite din valorile de prezis asociate cu grade de încredere aferente. Este o tabela mapata direct, indexata prin adresa instructiunii Load (PC).
- ⇨ **LCT** (Load clasification table) – mecanism care identifica cu acuratete instructiunile Load predictibile. Utilitatea "*Load Value Prediction*" este evidentiata doar în cazul predictionarii corecte, în caz contrar determinând atât hazarduri structurale cât si cresterea latentei de executie a instructiunilor Load. Clasificarea se bazeaza atât pe *comportamentul dinamic* cât si pe *predictia anterioara* a respectivelor instructiuni Load rezultând urmatoarele grupuri generale de instructiuni: *nepredictibile*, *predictibile* si *puternic predictibile* sau *constante*. Prin tratarea separata a fiecarui grup de instructiuni în parte este posibila exploatarea avantajului maxim în fiecare caz: putem evita costul unei predictii gresite prin identificarea Load-urilor nepredictibile sau se poate evita timpul necesar accesului la memorie identificând si verificând Load-urile puternic predictibile. LCT reprezinta o tabela mapata direct de numaratoare saturate pe *n* biti, indexata cu cei mai putin semnificativi biti ai adresei instructiunii Load (PC_{low}). Desi mecanismul LCT identifica cu acuratete tipul de predictie aferent fiecarui Load (nepredictibil, predictibil, puternic predictibil) este necesara verificarea corectitudinii valorii generate de LVPT. Pentru Load-urile predictibile, se compara valoarea prezisa cu valoarea actuala existenta în sistemul ierarhic de memorie (vezi figura 6.13) iar pentru cele puternic

predictibile este utilizata structura de verificare CVU. Statistici serioase realizate pe sistemele Power PC si Alpha AXP, utilizând benchmarkuri complexe (*jpeg, compress, perl, xlisp, hydro2d, tomcatv* etc.) arata ca peste 80% din instructiunile Load predictibile/nepredictibile sunt clasificate corect, ca atare, de catre unitatea LCT.

⇨ **CVU** (Constant Verification Unit) – este o unitate de verificare utilizata în cazul Load-urilor puternic predictibile, pentru evitarea accesarii complete a sistemului conventional de memorie, prin fortarea intrarilor LVPT care corespund Load-urilor respective sa ramâna coerente cu memoria principala. Pentru intrarile tabelei LVPT clasificate puternic predictibile de catre LCT adresa datelor si indexul în tabela LVPT sunt plasate într-o tabela complet asociativa în interiorul CVU. Aceasta tabela este mentinuta coerenta cu memoria centrala prin invalidarea oricarei intrari care are adresa datei identica cu cea a unei instructiuni Store ulterioare (similar aici cu anteriorul concept de reutilizare). La executia unei instructiuni Load puternic predictibila (bazat pe starea curenta a automatului corespunzator din LCT) are loc o cautare dupa continut (adresa datei si indexul în LVPT concatenate) în tabela asociativa. Daca rezultatul este cu hit, avem garantia ca, valoarea (prezisa) aflata la intrarea LVPT este coerenta cu memoria principala, întrucât orice actualizare a memoriei, aparuta dupa ultima salvare a datelor ar fi invalidat intrarea corespunzatoare din CVU. Daca nici o intrare în CVU nu satisface conditia de cautare (miss), instructiunea Load îsi modifica statutul din "puternic predictibila" în acela de "predictibila", iar valoarea prezisa este comparata cu valoarea actuala citita din ierarhia conventionala de memorie.

Figura 6.13 prezinta mecanismul LVP si modul de interactiune între structurile hardware implicate (LVPT, LCT si CVU) la executia instructiunilor cu referire la memorie.

La citirea unei instructiuni Load, - din cache sau memoria centrala - cu cei mai putin semnificativi biti ai adresei instructiunii Load (PC_{LOW}) se adreseaza în paralel tabelele LVPT si LCT. Structura din urma, similar unei tabele de istorie a salturilor, determina daca va fi facuta sau nu o predictie, în timp ce LVPT, analog unei tabele cu destinatiile instructiunilor de salt, înainteaza valoarea prezisa. Aceasta va fi preluata prin bypassing de catre instructiunile dependente aflate în asteptare în statiile de rezervare. Odata generata adresa de acces la memorie, pe nivelul EX1 al structurii pipe, cache-ul de date si CVU sunt accesate în paralel. La returnarea datei reale din cache, aceasta este comparata cu valoarea prezisa, instructiunile dependente executate speculativ fie urmeaza parcursul normal - nivelul Write Back al structurii pipe - fie sunt retrimise spre executie, iar structurile

LVPT si LCT sunt actualizate în consecinta. Întrucât cautarea în CVU nu se realizeaza la timp pentru a preveni initierea accesului la memorie (structura CVU asociativa) singura data când se poate realiza totusi acest lucru este atunci când apar conflicte la bank-uri sau accese cu "miss" în cache. Avantajul rezultat astfel îl reprezinta reducerea numarului de blocuri conflictuale la primul nivel de cache. În oricare din cazuri, un "hit" în CVU va întrerupe încercarile succesive sau procesul de miss în cache. În timpul executiei unei instructiuni Store, se realizeaza o cautare complet asociativa dupa adresa respectivei instructiuni si toate intrarile cu hit sunt înlaturate din CVU.

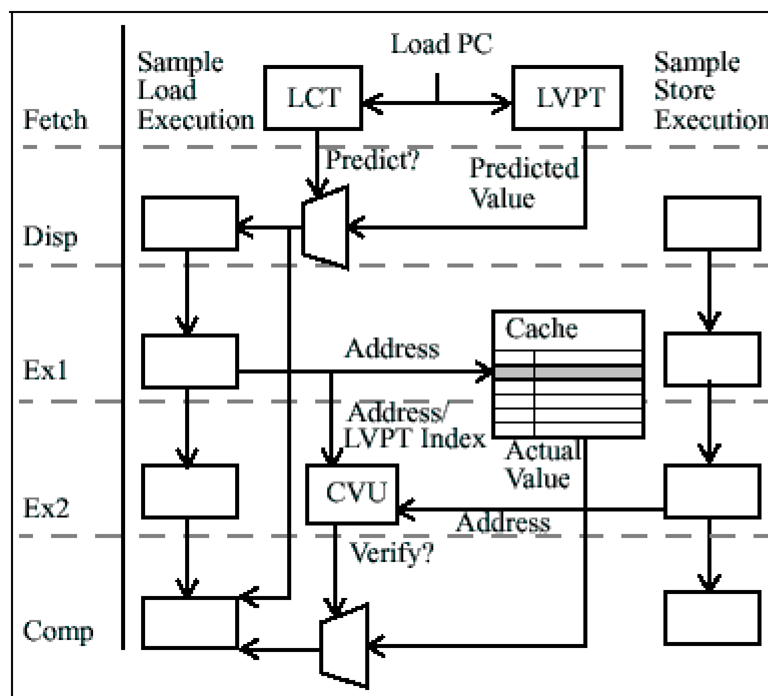


Figura 6.13. Schema bloc a mecanismului LVP

Load Value Prediction, spre deosebire de alte tehnici speculative, cum ar fi mecanismul de aducere anticipata a instructiunilor sau predictia ramificatiilor în program, reduce si nu creste, necesarul de largime de banda al memoriei. De asemenea, disponibilitatea foarte devreme (la începutul fazei de aducere a instructiunii IF) a indecsilor de accesare a tabelelor LVPT si LCT (accesul la respectivele tabele putând fi pipelineizat peste doua sau mai multe nivele pipe), complexitatea relativ redusa în proiectare si realizarea de tabele relativ mari fara a afecta perioada de tact a procesorului

sunt caracteristici care fac tehnica LVP atractiva pentru proiectantii de arhitecturi.

Cercetarile ulterioare care privesc predictia valorilor sunt canalizate pe extinderea LVP pentru predictia unei plaje de valori pentru o instructiune Load, ceea ce implica un multithreading speculativ în spatiul valorilor, sau executia speculativa utilizând valori generate de instructiuni altele decât Load. Pentru îmbunatatirea acuratetii de predictie a valorilor au fost propuse câteva tehnici. Acestea includ predictoare bazate pe istorie, hibride [46] si predictoare bazate pe context [47]; ele lucreaza la nivel de o singura instructiune si încearca sa prezica viitoarea valoare care va fi produsa de instructiune bazându-se pe valorile anterior generate. Întrucât respectivele scheme încearca sa înmagazineze o cât mai mare istorie de predictie, aceasta implica tabele hardware de mare dimensiune si cost ridicat de implementare. De asemenea, studiile cercetatorilor [48] arata ca, utilizând suportul compilatorului pentru extinderea predictiei valorilor si reutilizare dinamica la nivel de *basic block* se obtine o îmbunatatire substantiala a performantei procesoarelor (de la 1% la 14%, teste realizate pe benchmark-urile SPEC '95). Studii recente [55] abordeaza problematica cercetarii localitatii valorilor în contextul predictiei instructiunilor tip Store (memorare), cu implicatii deosebit de favorabile asupra performantei sistemelor uniprocesor dar mai ales multimicroprocesor (în special prin reducerea traficului prin reteaua comuna de interconectare).

Dupa prezentarea celor doua tehnici de exploatare a redundantei din program prin eliminarea dependentelor de date, IR (*instruction reuse*) si VP (*value prediction*), evidentiem succint, bazat pe [49], diferentele existente precum si modalitatile diferite de abordare a problemei reducerii caii critice de executie a instructiunilor într-un program.

VP predictioneaza rezultatele instructiunilor bazat pe rezultatele cunoscute anterior, efectueaza operatiile folosind valorile prezise iar executia speculativa este confirmata la un moment ulterior - *late validation* - când valorile corecte devin disponibile, deci dupa executia instructiunii. În cazul unei predictii corecte, calea critica este redusa întrucât instructiunile care s-ar executa secvential în mod normal, pot fi executate în paralel. În caz contrar, instructiunile executate cu intrari eronate trebuiesc reexecutate.

IR, pe de alta parte, recunoaste un lant de instructiuni dependente executat anterior si nu-l mai executa din nou - *early validation* - actualizând doar diferite date în tabelele hardware aferente. Astfel, IR *comprima* un lant de instructiuni din calea critica de executie a programului.

Figura 6.14 prezinta comparativ implementarea în structura pipeline a unei microarhitecturi a celor doua mecanisme: (a) - Value Prediction si (b) - Instruction Reuse.

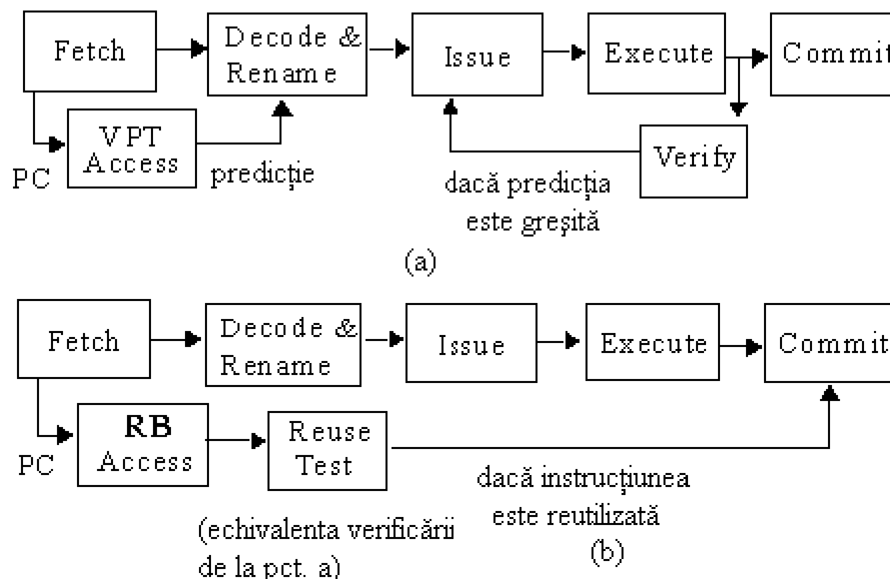


Figura 6.14. (a) Pipeline cu VP, (b) Pipeline cu IR

VP capteaza mai multa redundanta din program decât IR. Deoarece IR valideaza rezultatele devreme în pipe, bazat pe intrari, pot apare urmatoarele situatii dezavantajoase: daca intrarile unei instructiuni nu sunt disponibile în momentul realizarii testului de reutilizare atunci respectiva instructiune nu va fi reutilizata. O instructiune care produce un rezultat identic cu unul anterior, dar cu intrari diferite (operatii logice, instructiuni Load), nu va fi reutilizata. În schimb, VP poate realiza predictii corecte în ambele situatii prezentate mai sus întrucât valorile prezise nu depind de disponibilitatea intrarilor în pipe, si nici de importanta faptului ca ele sa fie sau nu identice cu instantele anterioare ale aceleiasi instructiuni.

Cele doua tehnici interactioneaza diferit cu mecanismul de branch prediction. IR reduce penalitatea datorata unei predictii gresite a salturilor din doua motive. În primul rând, când un branch predictionat gresit este reutilizat, predictia eronata este detectata mai devreme (în faza de decodificare) decât s-ar realiza daca saltul s-ar executa. Al doilea motiv îl constituie convergenta codului în programe. Astfel prin posibila reutilizare a codului, existent dupa punctul de convergenta a cailor de executie, si care este evacuat în cazul unei predictii eronate, tehnica IR îmbunatateste timpul de executie al programelor. Pe de alta parte, VP poate creste penalitatea introdusa de un branch gresit predictionat din urmatoarele motive: cauzeaza predictii eronate suplimentare si prin întârzierea rezolvarii saltului respectiv

(asteptându-se ca operanzii sai sa devina disponibili - dupa calculul si verificarea acestora).

VP si IR influenteaza concurenta asupra resurselor prin schimbarea atât a paternului în care resursele sunt folosite cât si a cererii efectuate. Prin colapsarea dependentelor de date cele doua tehnici determina executia mai devreme a instructiunilor. Întrucât o instructiune reutilizata nu se executa, IR tinde sa reduca concurenta la resurse. Sunt eliberate astfel resurse si puse la dispozitia altor instructiuni concurente. VP determina cresterea cererii de resurse întrucât instructiunile care se executa cu operanzi eronati se vor reexecuta. Executia acestor instructiuni poate fi reluata de mai multe ori daca valorile sunt predictionate gresit în mod repetat.

Impactul tehnicilor IR si VP asupra latentei de executie a instructiunilor este de asemenea antagonist. IR scade latenta de executie a operatiilor individuale (de la posibil mai multi cicli) la doar un ciclu (latenta de reutilizare a unei instructiuni). În schimb, VP nu scurtcircuiteaza executia - instructiunile trebuind sa se execute pentru a verifica predictia. Timpul total de executie a unei instructiuni va fi limitat astfel de latenta sa de executie (si verificare). Câstigul aici consta în faptul ca se deblocheaza instructiuni dependente prin procesul de predictie a valorii.

În contextul urmatoarei generatii arhitecturale de microprocesoare de înalta performanta, cea de a 4-a, se întrevede de asemenea implementarea unor mecanisme de aducere de tip Out of Order a instructiunilor, în plus fata de cele deja existente în executia instructiunilor. Astfel de exemplu, în cazul unei ramificatii dificil de predictionat, pe durata procesului de predictie, procesorul poate sa aduca anticipat instructiunile situate începând cu punctul de convergenta al ramurilor de salt. Aceste instructiuni fiind independente de conditia de salt, pot fi chiar lansate în executie. Când predictia se va fi realizat sau pur si simplu când adresa destinatie a ramificatiei va fi cunoscuta, procesorul va relua aducerea instructiunilor de la adresa destinatie a ramificatiei.

În viitorul apropiat, unitatea de executie va trebui sa lanseze spre unitatile functionale între 16 si 32 instructiuni în fiecare tact. Evident, executia instructiunilor se va face Out of Order, pe baza dezvoltarii unor algoritmi de tip Tomasulo (spunem "de tip Tomasulo" pt. ca acest algoritm este Out of Order doar în executie nu si în procesul de *dispatch* - lansare a instructiunilor din buffer-ul de prefetch înspre statia de rezervare; acest din urma proces este de tip In Order, lucru putin sesizat în cartile de specialitate care considera algoritmul ca arhetip al executiei Out of Order, fara nici o nuanta) [50]. Statiile de rezervare aferente unitatilor de executie, vor trebui sa aiba capacitati de peste 2000 de instructiuni Pentru a evita falsele dependente de date (WAR, WAW), procesoarele vor avea mecanisme de

redenumire dinamica a registrilor logici. Desigur, tehnicile de scheduling static vor trebui îmbunatatite radical pentru a putea oferi acestor structuri hardware complexe suficient paralelism Se estimeaza atingerea unor rate medii de procesare de 12-14 instr. / tact, considerând ca se pot lansa în executie maximum 32 instr. / tact. La ora actuala, cele mai avansate procesoare, cu un potential teoretic de 6 instr. / tact, ating în realitate doar 1.2-2.3 instr. / tact [51].

Aceste rate mari de procesare, impun executia paralela a cca. 8 instructiuni Load/ Store. Aceasta implica un cache de date primar de tip multiport si unul secundar, de capacitate mai mare dar cu porturi mai putine. Miss-urile pe primul nivel, vor accesa al 2-lea nivel. Pentru a nu afecta perioada de tact a procesorului, este posibil ca memoria cache din primul nivel sa fie multiplicata fizic.

Aceste noi arhitecturi, care executa trace-uri ca unitati de procesare, vor putea permite procesarea mai multor asemenea trace-uri la un moment dat, ceea ce conduce la conceptul de procesor multithreading. Asadar paralelismul la nivel de instructiuni (ILP- Instruction Level Parallelism) va fi înlocuit cu unul mai masiv, constituit la nivelul thread-urilor unei aplicatii (TLP- Thread Level Parallelism). În acest scop, arhitectura va trebui sa contina mai multe unitati de procesare a trace-urilor, interconectate. La aceasta, se adauga o unitate de control "high-level", în vederea partitionarii programului în thread-uri de instructiuni independente. Se poate ajunge astfel la o rata de procesare de mai multe trace-uri / tact fata de instructiuni / tact, metrica de performanta obisnuita a procesoarelor superscalare actuale. E posibil ca aceste TLP-uri sa acopere "semantic-gap"-ul existent între paralelismul la nivel de instructiuni si respectiv cel situat la nivelul programelor, mult mai masiv.

Exemple recente de arhitecturi bazate pe executie speculativa si multithreading includ: arhitectura multiscalara [37], trace procesorul [52], arhitectura superthread [53], procesorul superspeculativ. Procesorul superthread reprezinta un hibrid între arhitecturi multithreading si superscalare, care speculeaza dependentele de control (ramificatii conditionate) iar cele de date le rezolva dinamic. Trace procesorul si procesorul superspeculativ speculeaza atât dependentele de date cât si cele de control, în timp ce arhitectura multiscalara este sustinatoarea unei abordari multithread cu expediere vasta de fire spre executie. O alta paradigma noua, oarecum asemanatoare, dar mai îndepartata probabil ca realitate comerciala, o constituie multiprocesoarele integrate într-un singur circuit, ca solutie în vederea exploatarii paralelismului masiv ("coarse grain parallelism"). Aceasta este încurajata si de anumite limitari tehnologice care ar putea afecta dezvoltarea arhitecturilor uniprocesor.

Un “*procesor multithread*” (PMT) detine abilitatea de a procesa instructiuni provenite din thread-uri (fire de executie) diferite fara însa a executa pentru aceasta schimbari de context (*context switches*). PMT gestioneaza o lista a thread-urilor active si decide într-o maniera dinamica asupra instructiunilor pe care sa le lanseze în executie. Coexistenta mai multor thread-uri active permite exloatarea unui nou tip de paralelism numit “*Thread Level Pararlelism*” (TLP). Instructiunile din thread-uri diferite, fiind independente între ele, se pot executa în paralel ceea ce implica grade superioare de utilizare ale resurselor precum si mascarea latentei unor instructiuni. În acest ultim sens, de asemenea, gestiunea branch-urilor este simplificata, latenta acestora putând fi (partial) acoperita de instructiuni apartinând unor thread-uri diferite si deci independente de conditia de salt. De asemenea, efectul defavorabil al miss-urilor în cache poate fi contracarat prin acest multithreading (daca un thread genereaza un miss, CPU poate continua procesele de aducere ale instructiunilor din cadrul celorlalte thread-uri).

Desi multithreadingul îmbunatateste performanta globala, se cuvine a se remarca faptul ca viteza de procesare a unui anumit thread nu se îmbunatateste. Mai mult, este de asteptat chiar ca viteza de procesare pentru fiecare thread în parte sa se degradeze întrucât resursele trebuiesc partajate între toate thread-urile active. Cu alte cuvinte, acest TLP se preteaza la a fi exploatat în modurile de lucru ale sistemelor de operare de tip multiprogramare sau multithread. Partajarea multiplelor resurse hardware în vederea implementarii mai multor “*contexte*” de procesare aferente fiecarui thread, implica probleme dificile în cadrul unui PMT (mecanism de aducere a mai multor instructiuni de la adrese necontigue, structuri de predictie multiple, lansarea în executie a mai multor instructiuni apartinând unor thread-uri distincte etc).

Simularea unor arhitecturi PMT devine extrem de sofisticata, clasicele benchmark-uri nemaifiind aici de nici un ajutor. Trebuie lucrat în medii software bazate pe multiprogramare ceea ce nu este deloc usor de implementat si mai ales simulat si evaluat.

Se estimeaza ca toate aceste idei arhitecturale agresive, sa poata fi implementate într-un microprocesor real, abia atunci când tehnologia va permite integrarea "on-chip" a 800 milioane -1 miliard de tranzistori, ceea ce va fi posibil în jurul anului 2010 (predictie "Semiconductor Industry Association" în 1996 [51]). La acest nivel de dezvoltare tehnologica va fi posibila de asemenea integrarea "on-chip" a memoriei DRAM, la un timp de acces de cca. 20 ns. Ideea este atragatoare pentru ca la aceeasi suprafata de integrare, o memorie DRAM poate stoca de cca. 30-50 de ori mai multa informatie decât o memorie SRAM pe post de cache. Se estimeaza ca într-o

prima faza, un DRAM integrat de 96 MB va necesita 800 milioane de tranzistori ocupând cca 25% din suprafata circuitului (vezi pentru detalii http://iram.cs.berkeley.edu/).

În orice caz, se pare ca pentru a continua si în viitor cresterea exponentiala a performantei microprocesoarelor, sunt necesare idei noi, revolutionare chiar, pentru ca în fond paradigma actuala este, conceptual, veche de circa 15-20 de ani. Ar fi poate necesara *o abordare integratoare*, care sa îmbine eficient tehnicile de scheduling software cu cele dinamice, de procesare hardware. În prezent, dupa cum a rezultat din cele prezentate pâna acum, separarea între cele 2 abordari este poate prea accentuata. În acest sens, programele ar trebui sa expliciteze paralelismul intrinsec într-un mod mai clar. Cercetari actuale arata ca un program optimizat static merge mai prost pe un procesor Out of Order decât pe unul In Order. Printre cauze se amintesc expansiunea codului dupa reorganizare, noile dependente de date introduse prin executia conditionata a instructiunilor, faptul ca instructiunile gardate nu permit executia Out of Order etc. Cu alte cuvinte, separarea schedulingului dinamic de cel static este o prejudecata nociva dar care este din pacate deja consacrata în ingineria calculatoarelor unde practic nimeni nu si-a pus problema dezvoltarii unui optimizator de cod dedicat unei masini cu procesare Out of Order. Cercetarea algoritmilor ar trebui sa tina seama cumva si de concepte precum, de exemplu, cel de cache, în vederea exploatarii localitatilor spatiale ale datelor prin chiar algoritmul respectiv. Cunoastem, la ora actuala, relativ putine lucruri despre ce se întâmpla cu un algoritm când avem implementate ierarhii de memorii pe masina fizica. În general, algoritmii nu tin cont la ora actuala de caracteristicile masinii si acest lucru nu este bun pentru ca algoritmul nu ruleaza într-un "eter ideal" ci, întotdeauna din pacate, pe o masina fizica având limitari importante (de ex. elementele unui tablou se pot afla partial în cache si partial pe disc!). Astfel, dihotomia teorie - practica devine una artificiala si chiar nociva. Desigur, asta nu înseamna ca programatorul va trebui sa devina expert în arhitectura computerelor, dar nu o va mai putea neglija total daca va dori performanta. În noua era “post PC” spre care ne îndreptam, centrata pe Internet si tehnologia WWW, fiabilitatea, disponibilitatea si scalabilitatea vor trebui sa devina criterii esentiale alaturi de performanta în sine, ceea ce implica iarasi necesitatea unei noi viziuni pentru arhitectul de computere.

De asemenea, se poate predictiona o dezvoltare puternica în continuare a procesoarelor multimedia. Aplicatiile multimedia spre deosebire de cele de uz general, nu impun în mod necesar complicate arhitecturi de tip superscalar sau VLIW. Aceste aplicatii sunt caracterizate de urmatoalele aspecte care le vor influenta în mod direct arhitectura:

- structuri de date regulate, de tip vectorial, cu tendinte de procesare identica a scalarilor componenti, care impun caracteristici de tip SIMD (*Single Instruction Multiple Data*), de natura vectoriala deci, a acestor procesoare;
- necesitatea procesarii si generarii raspunsurilor în timp real;
- exploatarea paralelismului la nivelul thread-urilor independente ale aplicatiei (codari / decodari audio, video, etc). De exemplu, una din aplicatiile rulate de noul procesor comercial Intel Pentium IV o reprezinta codarea în timp real a imaginilor provenite de la o camera digitala [54].
- localizare pronuntata a instructiunilor prin existenta unor mici bucle de program si nuclee de executie care domina timpul global de procesare.

Autorii acestei lucrari cred, ca o opinie personala, ca viitorul cercetarii în acest domeniu al microarhitecturilor avansate, consta intr-o abordare complet novatoare a problematicii arhitecturilor de calcul. Se impune deci, o abordare integratoare a acestor arhitecturi prin utilizarea unor tehnici si concepte diverse din stiinta si ingineria calculatoarelor. Exista înca un puternic spirit al specializarii înguste care face adesea ca paradigma domeniului respectiv sa fie una închisa în tipare preconcepute. Abordari recente arata însa ca sinergia unor instrumente aparent disjuncte ale stiintei calculatoarelor converge spre realizari novatoare ale unui anumit domeniu de cercetare. Ca exemplu, o idee novatoare pe care dorim sa o comunicam în acest sens este aceea de determinare automata a unor noi scheme de predictie a salturilor pe baza de algoritmi genetici. Aceasta abordare dezvolta automat scheme de predictie a branch-urilor integrate in arhitecturi superscalare pe baza unor algoritmi genetici, pornind de la o populatie initiala de predictoare cunoscute, descrise complet cu ajutorul unor arbori binari (de fapt, mai precis, printr-un “limbaj” special conceput in acest sens - *BPL: Branch Programming Language*). Dupa evaluarea acestor predictoare realizata prin simulare pe trace-uri de program, prin aplicarea unor operatori genetici (incrucisare, mutatie, inversie etc) asupra elementelor din populatia initiala, se construieste o noua generatie de predictoare obtinuta automat. Procedura anterioara se va repeta pana la obtinerea unor generatii de predictoare de mare performanta (din pacate si complexitate). Cu alte cuvinte, pornind de la un set initial de scheme cunoscute si descrise printr-un formalism propriu adecvat se vor dezvolta noi scheme aplicând operatori genetici de tip "crossover", mutatie, inversie, etc. Noii generatii de scheme i se va calcula rata de fittness (de fapt, acuratetea predictiei) pe baza unor simulatoare deja finalizate. Pe aceasta baza se vor determina in mod pseudoaleator, in conformitate cu algoritmii genetici, noile perechi de parinti si care vor produce noi structuri aferente

generatiei urmatoare, dar si acele structuri, în general mai putin performante, care vor fi eliminate din cadrul generatiei urmatoare. Mentionam ca stadiul actual al acestor cercetari este deosebit de promitator. In final, se doreste a se afla daca cele mai puternice predictoare obtinute prin programare genetica sunt fezabile a fi implementate in hardware. De asemenea, se doreste a se afla daca prin astfel de proceduri automate se pot determina predictoare genetice mai puternice decat cele proiectate de catre oameni. Evident ca o astfel de experienta se poate generaliza ulterior la alte tipuri de scheme. O alta contributie majora a acestui abordari consta in compararea schemelor clasice obtinute automat prin programarea genetica cu predictoarele neurale dezvoltate in premiera mondiala de grupul nostru de cercetare. De asemenea vom dezvolta cercetarile asupra predictoarelor neurale de branch-uri, mai ales in ceea ce priveste investigarea unor noi algoritmi de invatare statica sau/si dinamica. Mai mult, propunem pentru prima data o idee complet noua constand intr-un predictor neural, bazat pe un anumit tip de retea (LVQ- Learning Vector Quantization, MLP – Multilayer Perceptron cu invatare tip backpropagation etc.). Aceasta idee face o legatura surprinzatoare intre domeniul arhitecturii procesoarelor avansate si cel al recunoasterii formelor. Tot aici, intentionam sa dezvoltam predictorul neural de branch-uri in sensul inglobarii sale in compilator, prin urmare propunand un predictor static de ramificatii. Problema deschisa este in acest caz urmatoarea: ce caracteristici intrinseci semnificative ale corpurilor de program pot constitui intrari fezabile in reteaua neurala de predictie (taken / not taken) ? Scopul final va consta intr-o "pre-predictie" realizata prin compilator, care sa ajute mai apoi procesele de predictie dinamica pe care noi deja le avem implementate prin simulatoare specifice. In fine, vom compara rezultatele obtinute prin aceste procedee novatoare de predictie a ramificatiilor cu cele obtinute prin scheme clasice sau chiar mai putin clasice.

O contributie majora în elaborarea unor microarhitecturi de mare performanta, ar putea consta în cercetarea posibilitatii de integrare a acestor structuri de predictie într-o arhitectura superscalara care sa exploateze în mod agresiv reutilizarea dinamica a instructiunilor si/sau predictia valorilor. Tot aici, consideram ca utila investigarea posibilitatii integrarii sinergice a reutilizarii dinamice a instructiunilor cu arhitecturile de tip "trace cache". Este util credem, spre exemplu, a se cerceta posibilitatea rezolvarii simultane a "fetch bottleneck"-urilor (determinate în principal de instructiunile de tip "taken branch" care limiteaza largimea de banda a aducerii instructiunilor la 5-6) cu limitarile de tip "issue bottleneck" (determinate fundamental de gradul de secventialitate intrinsec programelor, mai precis de catre dependentele reale de date), ambele din pacate active pe

actualele arhitecturi superscalare. Remarcam o similitudine între problema predictiei valorilor si problema actuala, a predictiei adreselor destinatie aferente instructiunilor de salt indirect. Structurile de date implementate în hardware pentru ambele procese de predictie, au acelasi principiu de functionare si anume: asocierea cvasibijectiva a contextului de aparitie al instructiunii respective (Load sau Branch) cu data / adresa de predictionat, în mod dinamic, odata cu procesarea programului. Iata deci ca problematica predictiei în microprocesoarele avansate, tinde sa devina una generala si ca urmare implementata pe baza unor principii teoretice mai profunde si mai elevate, asa cum este de altfel normal. Aceasta are drept scop principal si imediat, executia speculativa agresiva a instructiunilor, cu beneficii evidente în cresterea gradului mediu de paralelism prin paralelizarea unor instructiuni provenite din basic-block-uri diferite.

O alta idee interesanta o constituie abordarea integratoare din punct de vedere hardware-software. Toate arhitecturile dezvoltate trebuie evaluate si optimizate prin simulari complexe si deosebit de laborioase, utilizând benchmark-uri reprezentative. Cu alte cuvinte trebuie determinata maparea optimala a structurii hardware din punct de vedere al performantei executiei unor programe considerate a fi “numitorul comun” în raportarea de performante. Astfel, pe aceasta baza de simulare, se vor putea determina si întelege aspecte subtile ale relatiei dintre anumite caracteristici intrinseci ale programelor (recursivitate, numeric/non-numeric etc.) si respectiv necesitatile si caracteristicile arhitecturii hardware. De asemenea ar trebui studiat în viitor impactul integrarii unor asemenea structuri de predictie în cadrul unor arhitecturi evolutioniste dar si revolutionare, precum cele bazate pe reutilizarea dinamica a instructiunilor. În opinia noastra, bazata pe o cercetare bibliografica laborioasa pe parcursul mai multor ani, cercetarile actuale în domeniul arhitecturilor ILP (Instruction Level Parallelism) sunt tributare unei abordari relativ conventionale, situata strict în sfera “arhitecturilor de calcul”. Desigur ca aceste cercetari, în special cele dezvoltate în universitatile americane, exceleaza în simulari si implementari laborioase, de mare performanta si acuratete, mai greu obtenabile în Romania azi. În schimb, metodologia cercetarii este una deja clasica, bazata pe simulari cantitative ale acestor arhitecturi. Relativ la acest context actual, dorim sa subliniem ca o abordare mai neconventionala, care sa utilizeze concepte ale unor domenii considerate pâna acum a nu avea legatura cu arhitectura sistemelor de calcul (retele neuronale, algoritmi genetici, algoritmi de predictie PPM etc.), poate sa genereze rezultate surprinzatoare, nebanuite chiar, precum si o îmbogatire a paradigmei arhitecturilor avansate. Astfel, este stiut ca în general arhitecturile si structurile hardware sunt concepute prin efortul si inspiratia creierului uman. Ca o alternativa la

aceasta abordare clasica, dupa cum am mai aratat, noi ne-am gandit sa contrapunem acestei metode cvasi-unice, o metoda de generare automata a unor asemenea structuri hardware, pornind de la o populatie initiala de structuri “clasice”, prin utilizarea algoritmilor genetici. De asemenea intentionam sa largim cercetarile cu privire la predictoarele neuronale pe care le-am introdus de curând si care fac din problema predictiei branch-urilor, în mod surprinzator, o problema de recunoastere a formelor (deci o problema de “hardware” devine de fapt una de inteligenta artificiala). Consideram deci ca printr-o astfel de abordare mai putin conventionala, putem contribui si noi, în mod real, la un progres semnificativ al cunoasterii în domeniul arhitecturilor de calcul paralele si neconventionale.

7. SIMULAREA UNEI MICROARHITECTURI AVANSATE

7.1. INTRODUCERE

Istoria procesoarelor contrapune doua paradigme pentru cresterea performantei, bazate pe software si respectiv pe hardware. Ideea ca arhitectura procesoarelor interactioneaza "accidental" cu domeniul software este complet gresita, între hardware si software existând în realitate o simbioza si interdependenta puternica, înca neexplorata corespunzator. În procesul de proiectare al procesoarelor, aferent generatiilor viitoare, accentul principal nu se mai pune pe implementarea hardware, ci pe proiectarea arhitecturii în strânsa legatura cu aplicatiile potentiale. Se porneste de la o arhitectura de baza (generica), puternic parametrizata, care este modificata si îmbunatatita dinamic, prin simulari laborioase pe programe de test (benchmark-uri) reprezentative. Procesoarele se proiecteaza odata cu compilatoarele care le folosesc iar relatia dintre ele este foarte strânsa: compilatorul trebuie sa genereze cod care sa exploateze caracteristicile arhitecturale, altfel codul generat va fi ineficient. Metodele de crestere a performantei arhitecturilor de calcul cu ajutorul compilatoarelor se numesc *statice*, pentru ca programul este analizat si optimizat o singura data, înainte de a fi lansat în executie, iar cele bazate pe hardware numindu-se *dinamice*, pentru ca sunt aplicate în timp ce programul se executa.

Concluziile actuale ale cercetatorilor în domeniul arhitecturilor de calcul sunt ca trei fenomene (viteza ceasului, integrarea pe o singura pastila si exploatarea paralelismului la nivelul instructiunii si al firelor de executie) contribuie la cresterea performantei totale a procesoarelor. Scopul general al unei cercetari privind arhitectura sistemelor monoprocesor, si implicit al simularii unei arhitecturi tip monoprocesor, îl reprezinta cresterea si evaluarea gradului de paralelism la nivelul instructiunilor existent în

programele de test, compilate pentru respectiva arhitectura (performanta depinzând foarte mult si de compilatorul folosit, mai precis de tehnicile de optimizare globala pe care acesta le implementeaza). Programele de test pe care le-au considerat autorii acestei lucrari, sunt exemple reprezentative de aplicatii de uz general, gândite sa manifeste comportamente similare cu scopul general al programelor de calculator (incluzând aplicatii grafice, multimedia, compilatoare, programe de sortare, compresie sunet si imagine, jocuri); unele dintre acestea au o natura puternic recursiva. Ele fac parte din suite de teste standardizate. Printre primele programe folosite în optimizarea procesoarelor de uz general se situeaza si benchmark-urile Stanford - propuse de profesorul John Hennessy în anul 1981 (numar maxim - 1 milion de instructiuni masina dinamice) si, mai nou, cele din seria SPEC (Standard Performance Evaluation Corporation, http://www.specbench.org) - aparute începând cu anul 1992 si variante de *upgrade* la aproximativ 3 ani (numar maxim - 2.5 miliarde de instructiuni masina dinamice). Dinamica domeniului a determinat ca simularile software actuale sa se realizeze inclusiv pe arhitecturi multithreading si multiprocesor, desigur utilizându-se benchmarkuri specifice (codari / decodari audio, video, etc). De exemplu, una din aplicatiile rulate de noul procesor comercial Intel Pentium IV o reprezinta codarea în timp real a imaginilor provenite de la o camera digitala.

Metodologia de simulare poate fi de doua tipuri:

❖ **Execution driven simulation** , caracterizata de cunoasterea în fiecare moment (ciclu "pipe") a continutului resurselor arhitecturale (registri, locatii de memorie, unitati functionale). Asadar, în acest caz, simularea se face foarte detaliat, la nivel de ciclu de executie al procesorului. Ca iesiri din acest simulator, rezulta diverse informatii utile precum: continutul resurselor, gradul de încarcare al acestora, rate de procesare, de hit etc. De asemenea, un simulator de acest tip poate sa genereze în final toate instructiunile masina ale programelor de test în ordinea în care se executa si sa le scrie în fisiere în locatii contigue într-un format gen: *Cod_operatie*, *Adr_curenta* si *Adr_destinatie*. Aceste fisiere rezultat se numesc trace-uri si pot fi deosebit de utile dupa cum se va arata în continuare. Adresa curenta reprezinta valoarea registrului Program Counter (al instructiunii curente) iar adresa destinatie reprezinta adresa de memorie a datei accesate, în cazul instructiunilor cu referire la memorie, sau adresa destinatie a saltului, în cazul instructiunilor de ramificatie. În general, fisierele trace contin doar instructiuni de tip: *branch*, *load* si *store* întrucât acestea sunt singurele care creeaza un impact asupra fluxului de control (branch) respectiv de date (load / store)

al programului. Instructiunile aritmetico-logice, relationale, de deplasare si rotire nu apar, în general, în aceste trace-uri.

- **Trace driven simulation** , metodologie asupra careia vom insista mai mult în actualul capitol, analizeaza secvential toate instructiunile din trace-urile generate de simulatorul bazat pe execution driven, cu scopul de a determina instanta optimala a arhitecturii numita procesor, ce urmeaza a fi implementata în hardware. Aceasta metodologie se preteaza la simularea cache-urilor de date si instructiuni, mecanismelor de memorie virtuala etc., datorita faptului ca ofera pattern-uri reale de adrese, în urma executiei unor programe reprezentative.

Etapele de simulare, comparare si determinare efectiva ale unei arhitecturi optime, pornind de la sursa HLL (High Level Languages) a programelor de test si pâna la implementarea hardware a arhitecturii determinate sunt evidentiate în figura urmatoare.

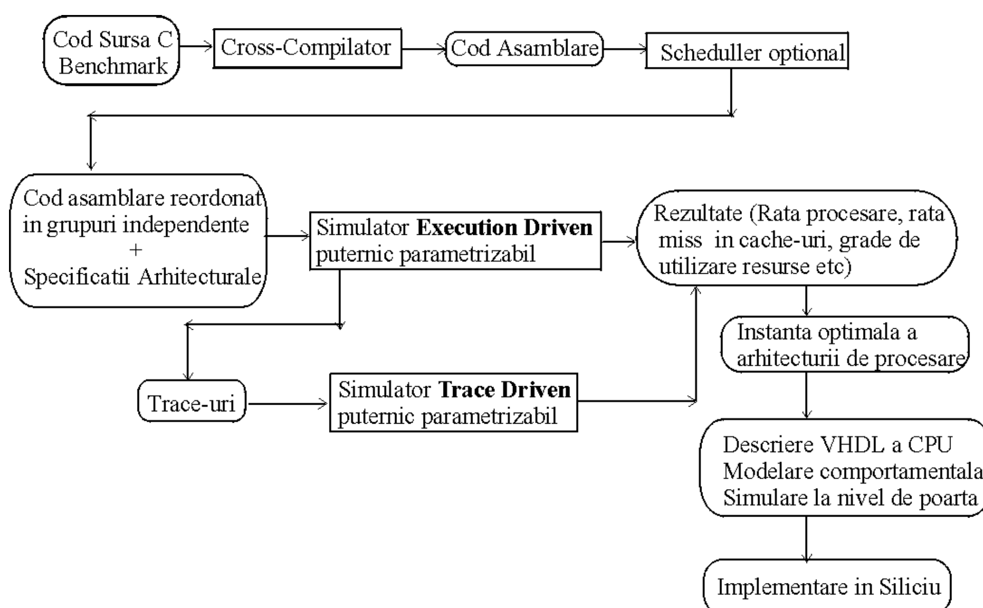


Figura 7.1. Etapele de simulare si determinare ale unei arhitecturi

Codul original sursa (în limbajul C aici) al benchmarkului este trecut întâi printr-un **cross-compilator** (de exemplu generatorul de cod obiect "gnuCC" sub Unix) care produce formatul corect al codului în mnemonica de asamblare, precum si directive de asamblare, comenzi de alocare a memoriei etc. Optional, codul rezultat în acest punct poate fi "*rearanjat*" prin intermediul unui **scheduler**, care împacheteaza instructiunile în grupuri de instructiuni independente. Codul obiect rezultat este trecut printr-un

simulator execution driven puternic parametrizabil (vezi paragrafele anterioare), care produce la iesire un fisier trace de instructiuni. Acesta reprezinta, asa cum am mai aratat, totalitatea instantelor dinamice, aferente instructiunilor statice (codul masina al benchmark-urilor în mnemonica de asamblare) scrise în ordinea executiei lor. În final, aceste trace-uri constituie intrari pentru **simulatorul trace driven** , de asemenea parametrizabil, care genereaza parametrii completi aferenti procesarii, pentru determinarea optimului de performanta în anumite conditii de intrare.

Capitolul de fata îsi propune sa fie un ghid software util sau / si deosebit de necesar tuturor acelora care doresc sau trebuie sa realizeze un simulator (masina virtuala) - util în evaluarea si optimizarea performantelor unei arhitecturi paralele de calcul - si nu stiu cum si cu ce anume sa înceapa. Este de dorit a nu se întelege ca varianta de simulator prezentata este singura solutie posibila de implementare, sau singurul mod de realizare al unui simulator software. Implementarea simulatorului în mediul Developer Studio[1] din Visual C++ s-a facut în primul rând datorita afinitatii si încrederii autorilor în acest mediu dar si bazat pe faptul ca limbajul C++ ofera un suport puternic pentru programarea orientata pe obiecte: mosteniri multiple, redefinirea operatorilor, functii si clase prieten etc. Conceptele de mostenire si polimorfism creeaza premisele dezvoltarii ulterioare (extinderii) a variantei actuale de simulator. De asemenea, implementarea trebuie realizata în asa maniera încât, orice modificare (adaugare) în hardware sau software sa fie facuta cu minim de efort. Privind din punctul de vedere al utilizatorului se considera imperios necesara o interfata vizuala prietenoasa, bazata pe meniuri, ferestre de dialog, imagini grafice edificatoare etc.

7.2. PRINCIPIILE IMPLEMENTARII SOFTWARE

Implementarea simulatorului s-a facut în limbajul Visual C++ 6.0, ultima versiune de compilator C++ existent la data scrierii codului aferent simulatorului (1999), produs de firma Microsoft pentru dezvoltarea de programe în mediul Windows. Pe lânga compilator, pachetul Visual C++ contine biblioteci, exemple si documentatia necesara pentru crearea

[1]*Mediul de dezvoltare integrata Developer Studio reprezinta componenta centrala a limbajului Visual C++.*

aplicatiilor în sistemele de operare Windows 9x, Windows 2000 sau Windows NT.

Mediul Developer Studio pune la dispozitia programatorului o serie de instrumente de dezvoltare, cum ar fi:

- ♦ Un editor integrat înzestrat cu tehnica "*drag and drop*" (tragere si plasare), precum si cu facilitati de evidentiere a sintaxei.
- ♦ Un editor de resurse, folosit la crearea resurselor Windows (imagini bitmap, pictograme, casete de dialog, meniuri).
- ♦ Un program integrat de depanare ce permite rularea programelor, depistarea erorilor, corectarea codului sursa, recompilarea si lansarea din nou a programului în depanare.
- ♦ Un sistem help on-line necesar în obtinerea informatiilor mai mult sau mai putin subtile (sensibile la context) pentru orice instrument folosit în mediul Developer Studio, informatii referitoare la limbajul C++, biblioteca MFC (Microsoft Foundation Class), interfata de programare Windows. Este bazat în mare masura pe exemple.

Pe lânga instrumentele de depanare, editare si creare a resurselor, mediul Developer Studio pune la dispozitia programatorului trei "*vrajitori*" utilizati la simplificarea dezvoltarii programelor Windows:

- ❖ **AppWizard** (vrajitorul aplicatiilor) – utilizat pentru crearea structurii de baza a unui program Windows.
- ❖ **ClassWizard** – folosit pentru definirea claselor într-un program creat cu AppWizard, manevrarea controalelor incluse în casete de dialog etc.
- ❖ **OLEControlWizard** – folosit la crearea cadrului de baza a unui control OLE. Un control OLE este un instrument personalizat care suporta un set definit de interfete si este folosit drept componenta reutilizabila [3].

În continuare, se prezinta etapele de realizare a unui program în Visual C++:

1. Crearea scheletului programului folosind AppWizard.
2. Crearea resurselor folosite de catre program.
3. Adaugarea claselor si functiilor de manevrare a mesajelor folosind ClassWizard.
4. Crearea nucleului functional al programului. Asupra acestei etape se va insista în paragrafele urmatoare (**referinta**), oferindu-se o prezentare detaliata.
5. Compilarea si testarea programelor, folosind programul de depanare integrat Visual C++.

7.2.1. INTERFATA CU UTILIZATORUL. CREAREA RESURSELOR.

Pentru început sunt furnizate detalii sumare despre resursele software folosite în cadrul programului în scopul obtinerii unei interfete vizuale prietenoase, simplu de utilizat, care sa permita utilizatorului manevrarea usoara a simulatorului, interpretarea si prelucrarea eficienta a rezultatelor (eventual transferul de rezultate în format text sau grafic spre alte utilitare de prelucrare - Microsoft Word Graph, Excel, PowerPoint, Internet etc).

La lansarea în executie a simulatorului (de exemplu: "*cache.exe*", simulatorul nostru destinat memoriilor cache integrate intr-o arhitectura superscalara) pe ecranul calculatorului gazda apare o fereastra înzestrata cu un **meniu** principal (vezi figura 7.2). Meniurile sunt o componenta esentiala a majoritatii programelor Windows, cu exceptia unor aplicatii simple, orientate pe casete de dialog, toate programele Windows oferind un tip de meniu sau altul. Aplicatia creata (simulatorul) este bazata pe meniuri simple, imbricate sau derulante flotante în functie de operatiile executate de / asupra elementelor arhitecturii specificate. Un meniu este o lista de mesaje de comanda care pot fi selectate si transmise unei ferestre. Pentru utilizator, un articol de meniu este un sir de caractere indicând o operatie care poate fi executata de catre aplicatie. Fiecare articol de meniu are un identificator utilizat pentru identificarea articolului, la directionarea mesajelor de fereastra sau modificarea atributelor respectivului articol de meniu.

O cerinta impusa interfetei cu utilizatorul o reprezinta *mnemonica* (litera subliniata), care trebuie sa apara în fiecare articol de meniu si la apasarea careia sa se selecteze articolul de meniu corespunzator. Împreuna cu tastele fierbinti (*hot keys*) – Ctrl + litera; Alt + litera; etc. – mnemonicile îsi dovedesc utilitatea atunci când sistemul de calcul dispune doar de tastatura nu si de mouse.

Meniurile pot fi create dinamic sau ca resurse statice care se adauga programului. Tratarea meniurilor este simplificata folosind clasa **CMenu** apartinând bibliotecii de clase MFC. Pentru fiecare aplicatie creata AppWizard genereaza o resursa de meniu, ce poate fi modificata în vederea stergerii sau inserarii de articole de meniu suplimentare la respectiva aplicatie.

O structura de meniu într-o faza primara de dezvoltare a simulatorului ar trebui sa contina optiunile: **Fisiere**, **Configurare**, **Executa**, **Memorie**, **Întrerupere**, **Ajutor**. Submeniul File cuprinde optiuni privind *Selectia si deschiderea unui program de test* (**Open Traces**), *Resetarea configuratiei arhitecturale* (**Reset All**), *Resetarea (închiderea) programului de test activ*

(**Anuleaza Benchmark**) - atât a fisierului cod masina cât si a trace-ului, *Închiderea simulatorului* si revenirea în sistemul de operare (**Exit**). Submeniul Configurare permite în primul rând *Stabilirea principalilor parametri ai arhitecturii* (mai putin date despre cache-uri) - rata de fetch a instructiunilor, rata maxima de lansare în executie - *issue,* dimensiune buffer de prefetch, tipul si numarul unitatilor functionale de executie etc - (**Unitati Executie**), *Activarea sau dezactivarea mecanismului de forwarding* (**Forwarding**), *Încarcarea* (**Load_Config**) respectiv *Salvarea* (**Store_Config**) unei configuratii arhitecturale prestabilite. Aceste facilitati permit trecerea rapida de la un model minimal la unul maximal din punct de vedere arhitectural, fara a altera parametrii cum ar fi latentele instructiunilor, latentele cache-urilor etc. Submeniul Executa trebuie sa cuprinda optiuni de *Lansare în executie* (**Start_Procesare**) - în mod continuu, respectiv pas cu pas, *posibilitatea de a vizualiza rezultatele* - în mod text (**Afisare Rezultate**) si respectiv în mod grafic (**Rezultate Grafice**) - în functie de simularea efectuata (**IR = f(FR)**). Optiunea de executie *Pas cu Pas* (Trace) permite examinarea starii interne a majoritatii elementelor componente ale arhitecturii de procesare (bufferul de prefetch, setul de registri generali, unitati functionale de executie). Alegând optiunea Memorie utilizatorul va trebui sa selecteze tipul cache-ului asupra carora se vor face modificari (**Instr_Cache**) respectiv (**Data_Cache**). Pe oricare dintre ramurile selectate utilizatorul va putea specifica arhitectura cache-ului (mapat direct, gradul de asociativitate), dimensiunea cache-urilor, dimensiunea blocului din cache-ul de date, penalitatea în caz de miss în cache, tipul de scriere (write back / write through), utilizarea sau nu a unui DWB (Data Write Buffer) care sa functioneze în paralel cu procesorul, preluând sarcina acestuia de a scrie în cache-ul de date (respectiv memorie în caz de miss). De asemenea, ar fi indicata prezenta unei optiuni în acest submeniu care sa permita vizualizarea, permanenta sau la anumite momente de întrerupere a executiei benchmark-ului, a datelor respectiv instructiunilor din cache. Submeniul Întrerupere permite *Stabilirea* (**Set**) si *Stergerea partiala* (**Reset**) *sau totala* (**Reset All**) a punctelor de întrerupere la executia fisierelor de test. Optiunea Ajutor trebuie sa contina explicatii si referinte detaliate despre arhitectura si functionarea simulatorului, resursele hardware si software utilizate, optiunile de meniu, facilitatile oferite de simulator, rezultatele generate.

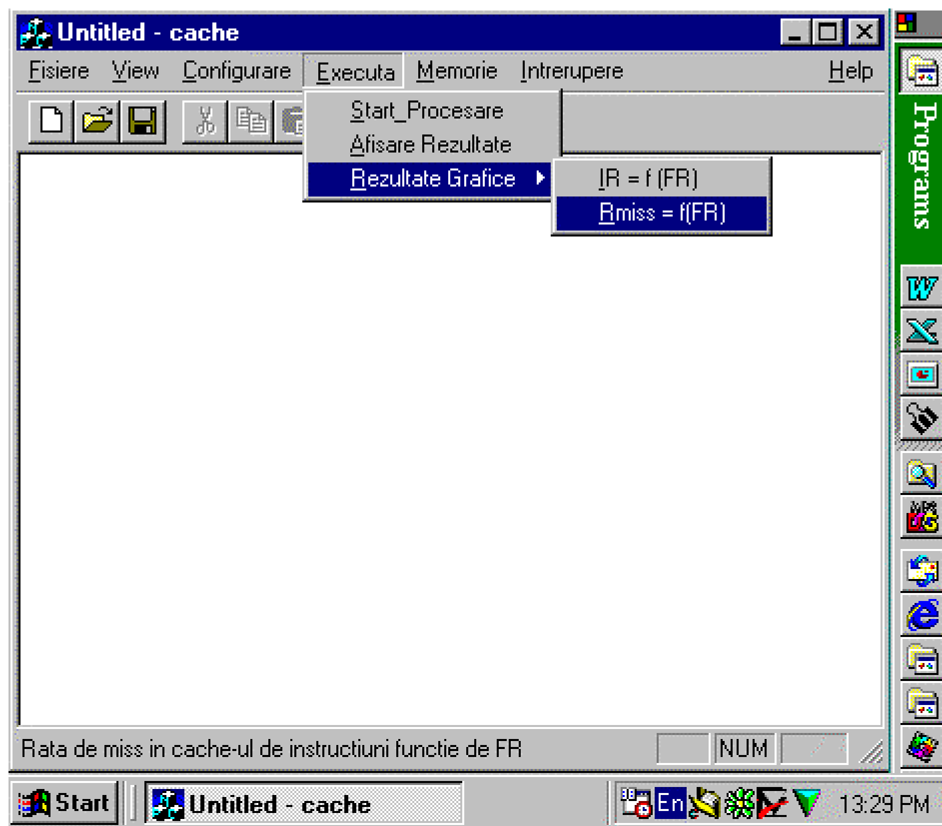


Figura 7.2. Meniul principal al simulatorului

O alta resursa software, frecvent utilizata la implementarea simulatorului o constituie **casetele de dialog**. Acestea reprezinta de fapt un tip specializat de ferestre. Deoarece sunt folosite cu precadere perioade scurte de timp, sunt de obicei stocate ca resurse de program si se încarca numai atunci când sunt necesare. Resursele program sunt memorate într-un fisier **.exe** - pe harddisc, dar se încarca numai când sunt efectiv necesare. Casetele de dialog sunt utilizate pentru a prezenta informatii si pentru a colecta date de intrare de la utilizator. Pot avea orice marime, variind de la casete simple de mesaj, care afiseaza o singura linie de text (Ex: "*Simularea s-a încheiat*!", "*Activare mecanism de forwarding. Simulatorul va fi resetat*!") - vezi figura 7.3, pâna la casete de dialog de mari dimensiuni, care contin controale sofisticate: de editare, de tip buton, de tip caseta lista sau caseta combinata, de desfasurare, de tip vizualizare lista, OLE - grila, grafic. În mod normal, casetele de dialog se folosesc pentru a concentra informatie si pentru a asigura reactia utilizatorului unui program.

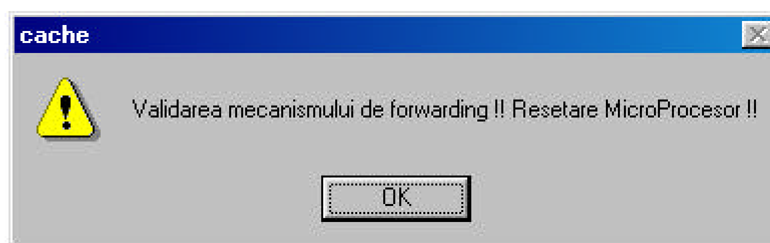


Figura 7.3. Caseta de dialog simpla

Tipul de caseta de dialog cel mai des folosit este *caseta de dialog modala* (caseta trebuie închisa înainte ca utilizatorul sa poata efectua o alta operatie), care contine de obicei mai multe controale utilizate pentru interactiunea cu un program. O caseta de dialog fara mod permite efectuarea altor operatii în timp ce caseta este deschisa (Ex: *Find and Replace* - utilizata de Developer Studio). Casetele de dialog se mai folosesc pentru comunicatia uni-sens cu utilizatorul ("ecranele de prezentare" utilizate la lansarea unui program pentru afisarea de informatii legate de copyright sau de pornire - vezi figura 7.4).



Figura 7.4. Comunicatie unisens prin casete de dialog

Casetele de dialog se mai folosesc pentru a înstiinta utilizatorul în privinta progresului unei operatiuni de durata (vezi figura 7.5). Situatia de mai jos prezinta stadiul procesarii benchmark-ului selectat dupa configurarea arhitecturii.

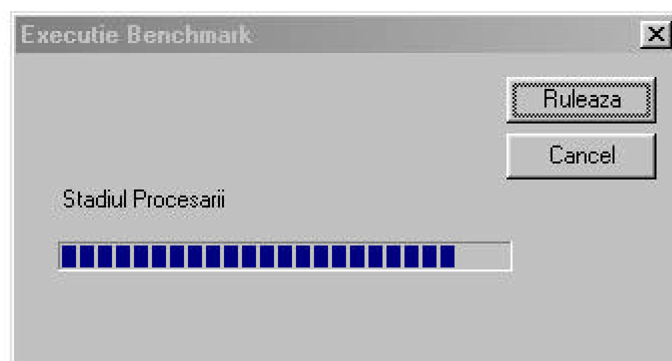


Figura 7.5. Caseta de dialog - *Executie Benchmark* - cuprinzând un control de desfasurare

Developer Studio faciliteaza utilizarea casetelor de dialog dintr-un program Windows. Toate etapele necesare procesului sunt automatizate, iar instrumentele utilizate pentru crearea casetelor de dialog si introducerea acestora într-un proiect sunt toate integrate. Developer Studio permite crearea si configurarea pe cale vizuala a unei casete de dialog. Controalele pot fi adaugate si dimensionate prin intermediul mouse-ului. Atributele casetei de dialog si a controalelor acesteia pot fi stabilite executând un simplu click de mouse. În general, adaugarea unei casete de dialog într-un program implica patru etape:

1. Proiectarea si crearea unei resurse caseta de dialog folosind instrumentele din Developer Studio.
2. Utilizarea facilitatii ClassWizard pentru a crea o clasa C++ derivata din **CDialog** care sa gestioneze caseta de dialog.
3. Adaugarea functiilor care sa trateze mesajele transmise casetei de dialog, daca este necesar.
4. În situatia în care caseta de dialog este selectata din meniul principal, resursa de meniu trebuie sa fie modificata, iar functiile de tratare a mesajelor create folosind ClassWizard.

În continuare se descriu câteva din cele mai utilizate casete de dialog în cadrul simulatorului. Efectuând una din urmatoarele 3 operatii: click pe pictograma Open (dosarul deschis) din bara de instrumente, apasând combinatia de taste Ctrl+O sau selectând Open Traces din meniul Fisiere, pe ecranul calculatorului apare o fereastra (caseta de dialog **Open**) care va facilita selectia si deschiderea unui program de test. Diferenta dintre aceasta caseta de dialog si cea care apare la selectia optiunii Open a meniului File din Windows (NT) Explorer este ca fisierul selectat nu va fi deschis în mod editare ci va fi deschis în background în vederea simularii. De asemenea, la

selectia unui fisier trace (*.trc, în cazul nostru) va fi deschis automat si corespondentul fisier în format cod masina (*.ins, la noi) în acelasi mod - pentru simulare (daca exista în respectivul director, în caz contrar se va afisa un mesaj pentru înstiintarea utilizatorului; simularea necesita ambele fisiere pentru a putea starta). Caseta de dialog Open contine un control tip vizualizare lista care permite vizualizarea fisierelor în diverse formate (raport, pictograma mica/mare, lista) din directorul selectat. De asemenea, un control de tip caseta lista specifica tipul fisierelor ce vor fi vizibile prin controlul vizualizare lista. Un control de editare permite introducerea de la tastatura (sau selectia prin dublu-click de mouse) a fisierului de test dorit. Butonul Open valideaza (realizeaza deschiderea efectiva a fisierului selectat) iar butonul Cancel anuleaza toate activitatile anterioare.

La selectia optiunii Unitati Executie din meniul Executa este lansata fereastra principala de configurare a parametrilor arhitecturii (rata de fetch, rata de issue, dimensiune buffer prefetch, etc). Caseta de dialog **Unitati Executie** cuprinde un numar mare de controale editare unilinie pentru colectarea datelor de intrare de la utilizator. La apasarea butonului OK, valorile aflate în controalele de editare sunt înscrise în variabilele de program (parametrii arhitecturii superscalare). În situatia în care nu au fost introdusi toti parametri, încercarea de startare a executiei instructiunilor (optiunea Start Procesare a meniului Executa) va fi zadarnica (pe ecran va apare o caseta de dialog de tip uni-sens cu mesajul "*Parametrii incompleti*! *Selectati optiunea de meniu Configurare*!"). La apasarea butonului Cancel configuratia arhitecturala este resetata, parametrii ramânând cu vechile valori, anterior stabilite sau nule.

O alta caseta de dialog utilizata - **Afisare Rezultate**) - (selectie Afisare Rezultate din meniul Executa) care prezinta prin intermediul controalelor de editare cele mai importante rezultate (rata de procesare, rate de miss în cache-uri, speed-up-ul obtinut prin implementarea diverselor tehnici - reutilizare dinamica a instructiunilor si predictia valorilor, rate de utilizare a cache-urilor, numar de instructiuni procesate, procentajul hazardurilor RAW din timpul total de executie etc).

O ultima caseta de dialog implementata si mai importanta (în viziunea autorilor) ilustreaza rezultatele obtinute (anterior amintite) sub forma grafica - **Rata de miss în Cache-ul de Instructiuni** - selectie Executa / Rezultate Grafice / Rmiss=f(FR) (vezi figura 7.6). Principala componenta a acestei casete de dialog o constituie controlul de tip ActiveX (element de interfata cu utilizatorul realizat în tehnologie ActiveX) cu ajutorul caruia reprezentam sub forma grafica rezultatele calculate în urma simularii - fara a fi necesar sa apelam la alte medii gen Microsoft Excel, MSWord Graph, MSAccess. ActiveX reprezinta o colectie de tehnologii si metode care permit resurselor

software sa interactioneze între ele, fara a cunoaste limbajul în care ele au fost create (utilizarea obiectelor de sine statatoare, gata create în cadrul altor resurse). ActiveX™ se bazeaza pe modelul obiectual al componentelor (COM). COM permite obiectelor sa fie functionale în cadrul altor componente sau al aplicatiilor gazda. Rezultatele si alti parametrii invarianti de intrare sunt transmisi ca intrari diverselor metode ale obiectului atasat controlului ActiveX. Cu ajutorul functiilor membre, respectivul obiect permite transformarea din valori numerice în coloane verticale ale unei harti grafice, sunt afisate limitele superioare ale respectivelor rezultate prin *grid-uri*. Fiecare coloana de rezultate este, în cazul concret al simulatorului nostru, aferenta unui benchmark, ultima coloana reprezentând-o media aritmetica sau armonica dupa caz. Suplimentar, la apasarea butonului Save este creat un fisier text (*rez.txt*) care contine rezultatele simulate pâna în momentul respectiv în urmatorul format.

	sort	**tree**	**matrix**	**bubble**	**queens**	**tower**	**perm**	**puzzle**	**Hmin**
FR=4	3.4333	6.0045	0.0506	0.0454	13.2642	0.0000	0.0267	7.6752	0.0000
FR=8	2.8733	8.1122	0.0572	0.0346	14.7214	0.0000	20.3828	7.7634	0.0000
FR=16	3.8167	9.3909	0.0630	0.0345	12.6638	0.0000	26.1302	6.9828	0.0000

Coloanele marcate cu 0.0000 sugereaza faptul ca respectivul benchmark nu a fost simulat si implicit nu a fost calculata media performantelor. La apasarea butoanelor OK respectiv Cancel este închisa caseta de dialog - rezultatele ramânând memorate în structurile software utilizate. Resetarea grafica a obiectului ActiveX se va face la selectia optiunii de meniu Reset All din Fisiere (când are loc de fapt si resetarea configuratiei arhitecturale si a programului de test).

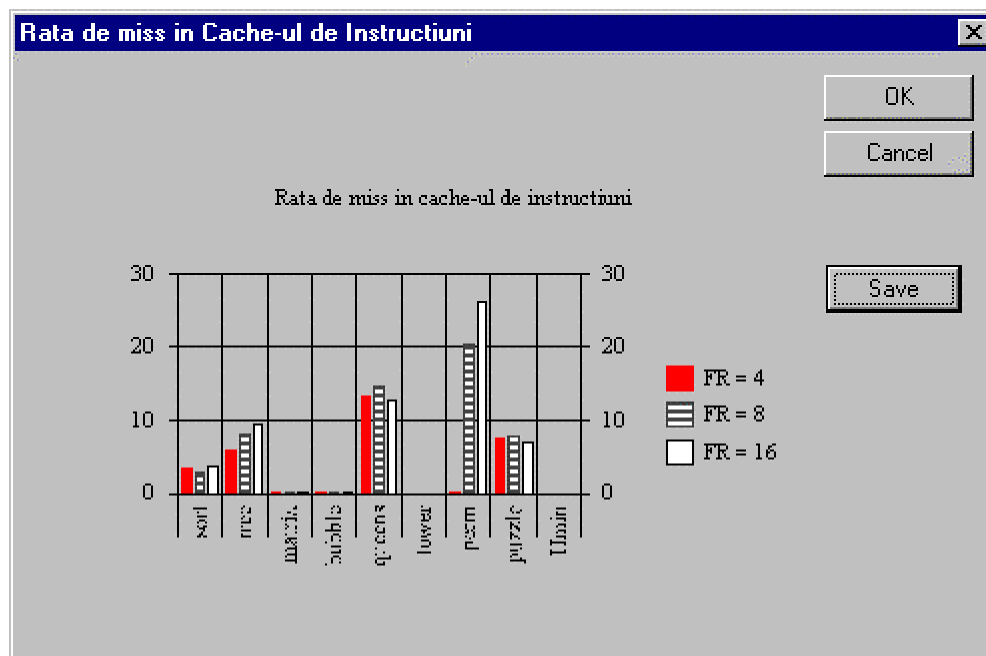


Figura 7.6. Afisarea rezultatelor sub forma grafica folosind controale ActiveX din Developer Studio

În continuare sunt enumerate si descrise sumar o parte din controalele utilizate în casetele de dialog anterior prezentate.

- Controalele de tip **buton** sunt poate cele mai flexibile tipuri de controale disponibile în Windows. Un buton este o fereastra de tip special care contine un text sau o eticheta bitmap si este localizata într-o caseta de dialog, bara de instrumente etc. Windows pune la dispozitie cinci tipuri de butoane, cele mai des utilizate fiind butoanele de apasare. Acestea prezinta un aspect în relief, tridimensional, care par apasate atunci când se executa clic cu mouse-ul asupra lor. Aproape fiecare caseta de dialog contine cel putin un control tip buton de apasare, folosit pentru a indica acele actiuni care pot fi comandate de utilizator. Între utilizarile cele mai comune ale unui buton de apasare se numara închiderea unei casete de dialog, începerea unei cautari sau solicitarea de asistenta. Reprezentarea majoritatii controalelor de tip buton se face cu ajutorul clasei **CButon**. Aproape toate controalele sunt activate în mod prestabilit, desi pot fi dezactivate initial prin setarea atributului corespunzator în lista de proprietati.
- În programele Windows colectarea datelor de intrare de la utilizator si afisarea textului si a datelor dinspre sau catre utilizator se realizeaza prin intermediul controalelor de **editare** (ferestre folosite pentru intrari de tip

text). Acestea pot fi de tip *unilinie* sau *multilinie* si sunt localizate în principal în casete de dialog dar si în orice alt context în care se doreste preluare sau afisare de date. Controalele de editare multilinie folosesc adeseori bare de derulare care permit introducerea unui volum de text mai mare decât poate fi afisat la un moment dat. Una din diferentele dintre controalele de editare si controalele de tip "buton de apasare" este aceea ca un control tip buton este folosit în mod normal pentru generarea de evenimente pe când controlul de editare desi poate genera evenimente este folosit cu preponderenta pentru stocarea datelor. O modalitate de interactiune cu un control de editare este prin intermediul unui obiect **CEdit** atasat controlului respectiv, folosind facilitatea ClassWizard. În mod prestabilit, un control de editare este gol când este afisat pentru prima data. Totusi, controalele de editare sunt adeseori folosite pentru afisarea informatiilor curente pentru utilizator, informatii care pot fi acceptate sau modificate. Altfel spus, un control de editare se poate folosi pentru a adresa o invitatie utilizatorului prin afisarea unei intrari prestabilite.

- **Caseta lista** este utilizata pentru a permite utilizatorului sa aleaga dntre un numar mare de optiuni. O caseta lista apare în mod normal într-o caseta de dialog, bara de controale etc. Este folosita pentru a include o lista de articole care pot fi selectate. Utilizatorul poate selecta articolele folosind tastatura sau executând clic pe un articol cu mouse-ul. Casetele lista pot fi cu *selectie simpla* (un singur articol selectat la un moment dat), sau cu *selectie multipla* (mai multe articole selectate la un moment dat). În mod prestabilit, casetele lista sunt cu selectie simpla. Selectia multipla poate fi utilizata în situatia în care se doreste simularea pe mai multe benchmark-uri la un moment dat (secvential sau în paralel - multithreading). Daca anumite articole nu pot fi afisate din motive de spatiu, casetele lista sunt prevazute cu o bara de derulare, pentru a facilita utilizatorului navigarea prin lista de articole. Lista de articole aferenta casetei lista este sortata (proprietate a listei ce poate fi exercitata de controlul, la fiecare inserare a unui articol, fara nici o interventie din partea programatorului). Casetele lista reprezinta cele mai simple controale care permit afisarea catre utilizator a unui numar arbitrar de articole. Sunt folosite adeseori pentru afisarea unor informatii extrase din baze de date sau rapoarte. Sortarea casetei lista înlesneste procesul de cautare si selectie al utilizatorului într-un numar foarte mare de articole. Clasa MFC **CListBox** poate fi folosita pentru gestiunea si interactionarea cu controlul tip caseta lista.
- Controalele de **desfasurare** sunt folosite pe scara larga pentru a indica evolutia unei operatii si se completeaza de la stânga la dreapta, pe masura

ce operatia se apropie de sfârsit. În Developer Studio sunt folosite pentru a indica desfasurarea unui proces de salvare sau încarcare a spatiului de lucru al unui proiect. Windows(NT) Explorer foloseste aceste controale la copierea sau mutarea fisierelor. Controalele de desfasurare constituie o modalitate utila de informare a utilizatorului despre stadiul unei operatii. În loc de a astepta o perioada nedefinita de timp, utilizatorul poate vedea ce fractiune din operatie mai trebuie finalizata. Gestiunea si interactionarea cu un control de desfasurare este realizata cu ajutorul clasei **CProgress**.

- Controalele tip **vizualizare lista** sunt controale extrem de flexibile, introduse pentru prima data de catre Windows '95. Sunt utilizate pentru afisarea de informatii însotite de pictograme asociate în patru formate diferite: raport, pictograma mica/mare, lista. La utilizarea unui control tip vizualizare lista, se poate folosi un meniu sau o alta metoda (bara de instrumente) pentru comutarea între diversele moduri de vizualizare. Atunci când permite utilizatorului comutarea între diversele stiluri de vizualizare, controlul tip vizualizare lista preda utilizatorului raspunderea cu privire la modul de afisare a informatiei. Controalele tip vizualizare lista accepta operatii de tragere si plasare, asemanator controalelor arborescente. Clasa MFC **CListCtrl** faciliteaza interactionarea cu controalele tip vizualizare lista si este asociata cu un control de acest tip folosind ClassWizard.

7.2.2. INTERFATA CU UTILIZATORUL. NUCLEUL FUNCTIONAL AL PROGRAMULUI.

În acest paragraf se va prezenta succint structura hardware a simulatorului dezvoltat special cu scopul de a furniza rezultate semnificative relative la performanta si structura optimala a memoriilor cache cu mapare directa, integrate într-o arhitectura superscalara parametrizabila. Trebuie retinut faptul ca structura simulatorului poate diferi de la o implementare la alta, în functie de ceea ce se urmareste (în situatia de fata obiectivul programarii îl reprezinta simularea interfetei procesor-cache pentru o arhitectura RISC superscalara parametrizabila, determinarea parametrilor optimali); cu toate acestea se va prezenta o microarhitectura generica de procesor care poate fi "îmbogatita" (extinsa) prin atasarea diverselor concepte sau tehnici de îmbunatatire a performantei, în functie de scopul urmarit.

Arhitecturile cu executie multipla a instructiunilor sunt compuse din doua mecanisme decuplate: mecanismul de aducere a instructiunilor (**fetch**) pe post de producator (activitate ce se desfasoara în ciclul **IF** – fetch instructiune – al procesarii pipeline) si respectiv mecanismul de executie a instructiunilor (**issue**) pe post de consumator (activitate ce se poate derula pe parcursul fazelor: executie / accesare memorie / scriere rezultat – functie de tipul instructiunii). Separarea între cele doua mecanisme (*arhitectura decuplata*) se face prin buffer-ele de instructiuni si statiile de rezervare (vezi figura 7.7). Instructiunile de ramificatie si predictoarele hardware aferente actioneaza printr-un mecanism de reactie între consumator si producator. Astfel, în cazul unei predictii eronate, bufferul de prefetch trebuie sa fie golit macar partial iar adresa de acces la cache-ul de instructiuni trebuie si ea modificata în concordanta cu adresa la care se face saltul.

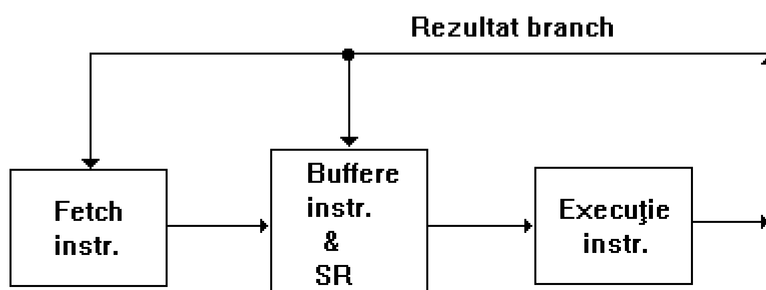


Figura 7.7. Architectura Superscalara Decuplata

Programul simulator dezvoltat, va procesa **trace-uri HSA** (Hatfield Superscalar Architecture) obtinute dintr-un simulator de tip *execution-driven*, special conceput la Universitatea din Hertfordshire, U.K. [1]. Acest simulator însa, nu abordeaza problema cache-urilor. Se vor determina într-un mod original, parametrii optimi de proiectare pentru acest tip de arhitectura cache. Simulatorul are în vedere atât cache-urile cu mapare directa mai pretabile în a fi integrate într-un microprocesor [4, 5] cât si cele având diferite grade de asociativitate, mai ales ca implementarea acestora pe scara larga în viitorul apropiat pare o certitudine.

Principalii parametri ai arhitecturii, alesi în concordanta cu nivelul tehnologic al ultimelor implementari, sunt urmatorii:

- **FR (rata de fetch)** - defineste marimea blocului accesat din cache-ul de instructiuni, mai precis numarul de instructiuni citite simultan din cache sau memorie (în caz de miss în cache) într-un ciclu de tact; poate lua valori de 4, 8 sau 16 instructiuni.

- **IBS (instruction buffer size)** - dimensiunea buffer-ului de prefetch, masurata în numar de instructiuni; plaja de valori: 4 (minim FR, altfel, nici o instructiune nu va putea fi plasata cu succes în buffer), 8, 16, 32, ≤ capacitatea cache-ului de instructiuni; buffer-ul de prefetch este o coada ce lucreaza dupa principiul FIFO (first in first out). Vor fi citite FR instructiuni simultan de la adresa specificata de PC (program counter) si depuse în partea superioara a buffer-ului. În acelasi ciclu de executie, instructiuni din partea inferioara sunt expediate spre unitatile de decodificare si executie. O intrare în buffer va contine câmpurile:

  OPCODE - codul operatiei executata de instructiunea respectiva;
  PC_crt - adresa (Program Counter-ul) instructiunii curente;
  DATE / INSTR - adresa din / la care se citesc / se scriu date din sau în memorie, în cazul instuctiunilor cu referire la memorie, respectiv adresa instructiunii tinta în cazul instructiunilor de salt.

- **capacitatea memoriilor cache** care variaza între 64 cuvinte si 16 Kcuvinte. Aceste capacitati relativ mici ale memoriilor cache se datoreaza particularitatilor benchmark-urilor Stanford utilizate. Acestea folosesc o zona restrânsa de instructiuni, limitata la cca. 2 Ko si o zona de date mai mare dar mai "rarefiata", care se întinde pe un spatiu de cca 24 Ko.

- **IRmax (issue rate maxim)** - numarul maxim de instructiuni, lansate în executie simultan într-un ciclu de executie, din buffer-ul de prefetch. Poate lua valorile: 2, 4, 8, 16 (maxim FR) instructiuni. Daca rata de fetch este mai mica decât numarul maxim de instructiuni executate concurent într-un ciclu, atunci performanta este limitata de procesul de *fetch instructiune*. Simulatorul implementat considera executia instructiunilor "*in order*" - ordinea initiala a instructiunilor. O instructiune va fi executata abia dupa ce toate celelalte instructiuni anterioare, de care ea depinde au fost executate.

- **Strategia de scriere în cache** : *write back* sau *write through*. Cu write through, informatia e scrisa atât în blocul din cache cât si în blocul din memoria principala. Prin write back informatia e scrisa doar în blocul din cache. Write back implica evacuare efectiva a blocului - cu penalitatile de rigoare - în memoria principala. Rezulta ca este necesar un bit *Dirty* asociat fiecarui bloc din cache-ul de date. Starea acestui bit indica daca blocul e *Dirty* (modificat cât timp a stat în cache), sau *Clean* (nemodificat). Daca bitul este "curat", blocul nu e scris la miss,

deoarece nivelul inferior – memoria principala - contine copia fidela a informatiei din cache. Daca avem citire din cache cu miss si Dirty setat pe ‘1’ atunci vom avea o penalizare egala în timp cu timpul necesar evacuarii blocului - existent în cache dar nu cel solicitat - la care se adauga timpul necesar încarcarii din memorie în cache a blocului necesar în continuare. La write through nu exista evacuare de bloc la cache-urile mapate direct, dar exista penalitati la fiecare scriere în memorie în lipsa unui procesor specializat de iesire (Data Write Buffer). Write through are de asemenea avantajul ca, urmatorul nivel inferior are majoritatea copiilor curente ale datei. Acest lucru e important pentru sistemele de intrare / iesire (I/O) si pentru multiprocesoare în vederea pastrarii coerentei variabilelor stocate în cache. Dispozitivele I/O si multiprocesoarele sunt schimbatoare: ele vor sa foloseasca write back pentru cache-ul procesorului si pentru a reduce traficul memoriei si vor sa foloseasca write through pentru a pastra cache-ul consistent cu nivelul inferior al ierarhiei de memorie. Oricare din cele doua strategii de scriere pot fi asociate cu cele de mentinere a coerentei cache-urilor în cadrul sistemelor multiprocesor, anume strategia "*write invalidate*", respectiv "*write broadcast*" (pentru detalii vezi paragraful 8.5.2).

- **Utilizare / neutilizare DWB (data write buffer)** - o coada FIFO de lungime parametrizabila, a carei valoare trebuie sa fie minim IRmax. Fiecare locatie contine adresa de memorie (virtuala) si data de scris. Consideram ca DWB contine suficiente porturi de scriere pentru a sustine cea mai defavorabila situatie (STORE-uri multe, independente si simultane în fereastra IRmax fiind deci posibile), oferind deci porturi de scriere virtuale multiple. În schimb D-Cache va contine un singur port de citire (LOAD) si un singur port de scriere (STORE), reflectând o situatie cât mai reala. Consideram latenta de scriere a datelor în DWB de 1 tact iar latenta de scriere a datelor în D-Cache este de 2-3 tacte (parametrizabila). Cu DWB sunt posibile deci STORE-uri simultane, fara el acestea trebuind serializate cu penalitatile de rigoare. În plus DWB va putea rezolva prin "bypassing" foarte elegant hazarduri de tip "LOAD after STORE" cu adrese identice, nemaifiind deci necesara accesarea sistemului de memorie de catre instructiunea LOAD.

Structura de pricipiu a arhitecturii superscalare tip Harvard (detine busuri si cache-uri separate pe spatiile de instructiuni si date) care a fost simulata este cea din figura (figura 7.8).

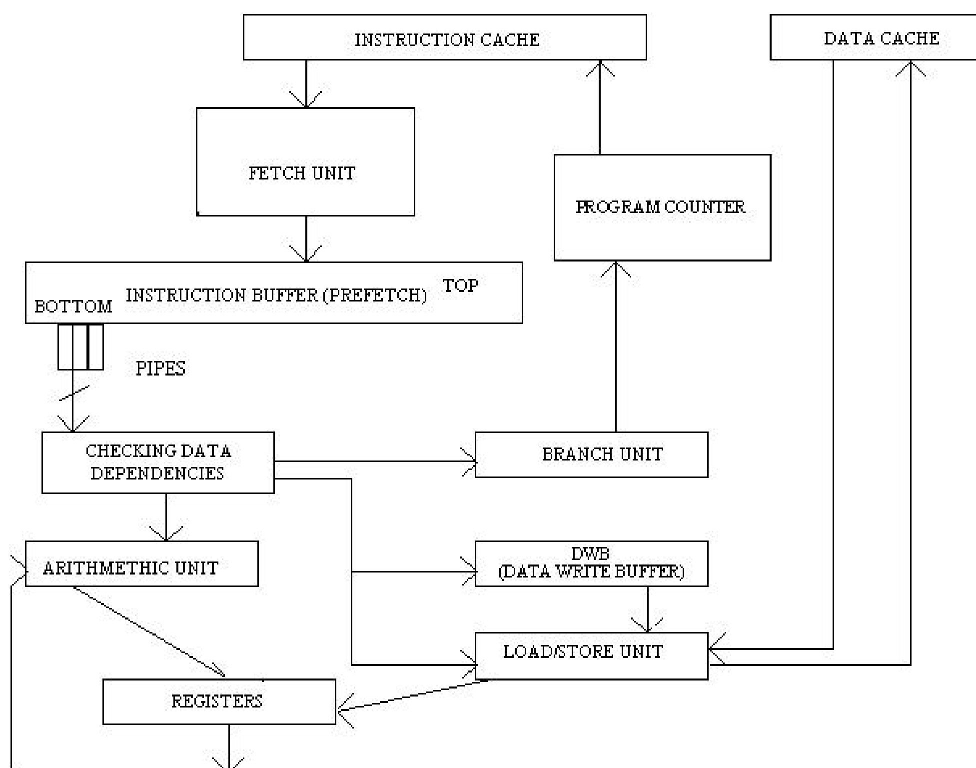


Figura 7.8. Schema bloc de principiu a arhitecturii superscalare Harvard simulate

Un cuvânt din memoria cache va contine doua câmpuri: un câmp de tag si un bit de validare V. Câmpul de tag contine blocul din memoria principala actualizat în cuvântul din cache de la indexul curent, iar bitul V valideaza informatia din cache în sensul în care, initial, fiecare cuvînt din cache va fi considerat invalid pâna la prima actualizare a sa din memoria principala. Memoria principala se acceseaza numai la miss în cache si va avea latente parametrizabile cuprinse între 10-20 tacti procesor (realist la nivelul performantelor tehnologiilor actuale). La miss în cache, trebuie deci introduse penalizari corespunzatoare în numarul tactilor de executie (în conformitate cu tehnica de scriere aplicata - *write back* sau *write through*). În cazul acceselor la spatiul de date, se introduc penalizari numai pentru instructiunile LOAD, în cazul instructiunilor STORE nemaifiind nevoie datorita procesorului de iesire specializat (DWB-Data Write Buffer) considerat pe moment idealizat si a bufferelor aferente. Asadar într-o prima faza a cercetarii scrierea s-a idealizat doar din motive legate de claritatea expunerii, oricum îngreunata de multitudinea caracteristicilor arhitecturale considerate. S-a considerat în mod realist ca este posibila executia simultana

a oricaror doua instructiuni cu referire la memorie cu restrictia ca adresele lor de acces sa fie diferite (cache biport pe date). Având în vedere ca se lucreaza pe trace-uri, aceasta analiza antialias perfecta este evident posibila. Întrucât simularea se face pe baza trace-urilor HSA ale benchmark-urilor Stanford, s-a presupus o predictie perfecta a branch-urilor în cadrul simularii. Simulatorul realizat trebuie sa elimine gâtuirile care limiteaza performanta si sa investigheze posibile schimbari (arhitecturale sau tehnici de optimizare) în scopul cresterii acesteia. Prin realizarea unui model de simulare detaliat pentru fiecare procesor, performanta obtinuta prin simulare este capabila sa asigure un rapid *feedback* în legatura cu schimbarile propuse.

Întrucât simularile autorilor s-au efectuat pe benchmark-urile Stanford vom descrie mai pe larg etapele de simulare, comparare si determinare efectiva ale unei arhitecturi optime, pornind de la sursa C a programelor de test si pâna la implementarea hardware a arhitecturii determinate. Cele 8 benchmarkuri Stanford scrise în C, au fost compilate special pentru arhitectura HSA utilizând compilatorul Gnu CC (sub Linux), rezultând programele obiect HSA. Acestea se întind pe un spatiu de cod mai mic de 600 instructiuni HSA iar spatiul de date utilizat nu depaseste 24 ko. Programele obiect HSA la rândul lor, au fost procesate cu un simulator special [1], care a generat trace-urile aferente (toate instructiunile masina în ordinea rularii lor, împreuna cu adresele acestora). Aceste trace-uri HSA sunt disponibile din simulatorul HSA [1] sub o forma speciala din motive de economie de spatiu pe disc (întemeiate la momentul realizarii simulatorului execution-driven în 1995) si ele contin practic doar instructiunile de LOAD (L), STORE (S) si BRANCH (B-numai cele care au provocat într-adevar salt), în ordinea procesarii lor în cadrul programului respectiv, fiecare cu PC-ul sau si cu adresa efectiva corespunzatoare (de salt - B sau de acces la data - L/S). Chiar si în aceasta forma, un asemenea fisier trace ocupa între 2-4 Mbytes, el continând practic câteva sute de mii de instructiuni ce au fost executate (între 72.000 si 800.000 de instructiuni masina în cazul celor utilizate aici).

Spre exemplificare se prezinta primele doua linii de cod din programul de test **fsort.trc** [1].

```
B  2   151      S 152 3968     B 153 120      S 121 3840     B 122 18
S 19  3712      S  20  3720    S 21  3724     B 22  4        S 6   6328
```

Prima instructiune din trace este <B 2 151> semnificând urmatoarele: PC-ul instructiunii de salt este 2, iar adresa urmatoarei instructiuni citite si ulterior executate este 151. Întrucât programul începe cu instructiunea al

carei PC=0, si aceasta nu exista în trace, rezulta ca primele doua instructiuni din program sunt aritmetico-logice. Secventa reala de instructiuni ar fi:

A 0 xxxx   A 1 xxxx   B 2 151.

Urmatoarea instructiune este cea de la adresa 151, dar cum ea nu se gaseste în trace, înseamna ca la aceasta adresa exista tot o instructiune aritmetica, iar PC_next (PC-ul urmatoarei instructiuni) este incrementat, neexistând nici o instructiune de salt care sa schimbe cursul programului. Instructiunea urmatoare având PC=152, este cu referire la memorie “Store la adresa 3968”. Urmeaza o noua instructiune de salt, PC_next devenind 120. La aceasta adresa întâlnim o noua instructiune aritmetica, urmata de un Store iar apoi un nou salt, samd.

Concomitent putem urmari în fisierul **fsort.ins** mnemonica în asamblare a instructiunilor citite si ulterior executate, trace-ul reprezentând cursul exact al programului - instructiune cu instructiune - în conditiile unui branch prediction perfect.

Prezentam desfasurarea - modul de citire si executie - al instructiunilor, în paralel, (trace si asamblare) a primelor doua linii din fisierul trace [2].

Fsort.trc	**fsort.ins**
A 0 xxxx	MOV GP, #4096
A 1 xxxx	MOV SP, #4096
B 2 151	BSR RA, _main (#0)
A 151 xxxx	SUB SP, SP, #128
S 152 3968	ST 0(SP), RA
B 153 120	BSR RA, _Quick (#0)
A 120 xxxx	SUB SP, SP, #128
S 121 3840	ST 0(SP), RA
B 122 18	BSR RA, _Initarr (#0)
A 18 xxxx	SUB SP, SP, #128
S 19 3712	ST 0(SP), RA
S 20 3720	ST 8(SP), R17
S 21 3724	ST 12(SP), R18
B 22 4	BSR RA, _Initrand (#0)
A 4 xxxx	SUB SP, SP, #128
A 5 xxxx	MOV R13, #74755
S 6 6328	ST _seed, R13

Tabelul 7.1.

Ilustrarea în paralel a executiei instructiunilor pe fisierele trace respectiv cod masina

Mai jos, se prezinta o descriere succinta a benchmarkurilor utilizate în cercetare.

Benchmark	**Total instr.**	**Descriere bench**
Puzzle	804.620	Rezolva o problema de "puzzle"
Bubble	206.035	Sortare tablou prin metoda "bulelor"
Matrix	231.814	Inmultiri de matrici
Permute	355.643	Calcul recursiv de permutari
Queens	206.420	Problema de sah a celor 8 regine
Sort	72.101	Sortare rapida a unui tablou aleator (quick sort)
Towers	251.149	Problema turnurilor din Hanoi (recursiva)
Tree	136.040	Sortare pe arbori binari

Tabelul 7.2.

Caracteristicile benchmark-urilor Stanford

Este acum evident, ca prin procesarea unui asemenea trace, având în vedere ca se dispune si de programul obiect HSA generat prin compilarea benchmarkului scris în C, se pot simula în mod realist, procesele legate de memoriile cache, predictorul hardware de ramificatii etc., integrate într-un procesor paralel. Implementarea simulatorului a fost facuta considerând urmatoarele:

- se aduc FR instructiuni din cache sau memorie în IB daca exista un spatiu disponibil mai mare sau egal cu FR; daca adresa urmatoare a unei instructiuni de salt (B) este una nealiniata cu FR, se va alinia si se va aduce noua instructiune multipla de la adresa aliniata, considerându-se astfel o implementare cache realista, în care accesele pot fi realizate doar la adrese multiplu de FR. Evident ca în acest caz, instructiunile inutil aduse, nu se vor lansa si contoriza în procesare;
- nu se pot aduce decât exact FR instructiuni din cache, din motive de aliniere adrese;
- executia instructiunilor se face In Order (o instructiune va fi executata abia dupa ce toate celelalte instructiuni de care ea depinde au fost executate), eludând deci complicatii hardware legate de modelarea si simularea unor algoritmi tip Out of Order [1, 6];
- nu pot fi lansate în executie decât maxim doua instructiuni cu referire la memorie în conditiile în care adresele lor de acces sunt diferite (antialias); utilizând un Data Write Buffer (DWB) este permisa din punct de vedere al procesorului lansarea unui numar maxim de instructiuni cu referire la memorie simultan în executie (IRmax) dar scrierea / citirea

efectiva în / din cache-ul de date, realizata de DWB, este limitata tot la doua instructiuni;
- daca apare miss în spatiul de instructiuni pe perioada "lunga" a accesarii memoriei principale, se vor executa în continuare instructiuni din IB.

Executia simulatorului se face în urmatorii pasi succesivi:

1. Solicitare parametri de intrare ai structurii superscalare (prin intermediul interfetei grafice de intrare anterior descrise) si anume: rata de fetch (FR), dimensiune IB, rata maxima de lansare în executie a instructiunilor din bufferul de prefetch (IRmax), capacitate cache-uri, tip cache (uniport / biport), tip arhitectura cache (mapat direct / grad asociativitate), nume fisier trace utilizat (*.trc), numar tacte penalizare la miss în cache (N) etc.

2. Initializare cache-uri (peste tot bit V - validare si TAG=0), initializare registru PC (Program Counter) cu adresa de start a programului de test, si initializare cu 0 a ceasului programului.

3. Lansare în executie a procesarii trace-ului.

Se va executa fetch instructiune, în ipoteza ca spatiul disponibil din IB o permite. Instructiunile vor fi citite din fisierul trace respectând logica de program. În caz de miss în cache, se penalizeaza timpul de executie si, daca este posibil, se vor executa în continuare instructiuni din IB.

Se verifica conflictele LOAD / STORE, sau eventualele hazarduri RAW între instructiunile aflate în "*fereastra curenta de instructiuni*" (egala cu IRmax) din IB în vederea executiei. Întrucât procesarea se face *in order* o dependenta RAW sau o ambiguitate a referintelor la memorie determina lansarea în executie a instructiunilor independente gasite pâna în acest moment, cu prima instructiune dependenta startând un alt grup de instructiuni posibil a fi lansate în paralel.

Sunt lansate simultan în executie maxim IRmax instructiuni independente iar ceasul programului este incrementat cu maximul dintre timpul consumat de procesul de *fetch* si respectiv cel de lansare în executie (*issue)*.

Simulatorul implementat genereaza urmatoarele rezultate, considerate ca fiind deosebit de importante pentru analiza propusa:
- numar de instructiuni procesate, numar total de tacte, rata medie de procesare (IR)
- rata de hit/ miss în cache-uri
- procentajul din timpul total cât IB este gol (stagnare practica a procesarii - IBE(%))

- procentajul fiecarui tip de instructiuni LOAD / STORE, ALU, BRANCH din trace
- determinarea carei metode de scriere în cache este mai viabila (Write Back vs. Write Through)
- procentajul din numarul tactelor cât exista alias-uri la memorie, procentajul hazardurilor RAW din timpul total de executie, etc.

Câstigul în performanta dobândit prin implementarea diverselor tehnici avansate de îmbunatatire a performantei procesoarelor (selective victime cache, reutilizare dinamica a instructiunilor, predictia valorilor, trace cache, etc).

În ciuda predictiei perfecte a branch-urilor, indicatorul IBE nu este trivial datorita posibilitatii aparitiei miss-urilor în cache-uri si latentelor ridicate ale memoriei principale (N), care pot conduce la golirea bufferului IB. Toate aceste rezultate sunt exprimate prin intermediul ferestrelor descrise în capitolul anterior dar si scrise într-un fisier (REZULT.SIM), permitându-se astfel prelucrarea lor ulterioara. Eventuale viitoare dezvoltari ale acestei cercetari pot avea în vedere implementarea predictorului hardware de branch-uri (dezvoltat tot de catre echipa de cercetare a autorilor si ajuns la varianta neuronala), modificarea tehnicii de lansare a instructiunilor în executie din *in order* în *out of order* sau chiar noi paradigme ale domeniului arhitecturii calcuatoarelor.

8. ARHITECTURA SISTEMELOR MULTIPROCESOR

8.1. DEFINIRI. CLASIFICARI

Sistemele multimicroprocesor (SMM) reprezinta sisteme de tip MIMD (Multiple Instruction Multiple Data), constituite din mai multe μprocesoare interconectate prin intermediul unei retele de interconectare (RIC) astfel încât sa permita programelor aflate în executie sa interschimbe date si sa-si sincronizeze activitatile.

De remarcat ca în cazul SMM exista în general mai multe programe în executie la un moment dat, bazat pe paralelismul spatial al hardware-ului. Pentru a exploata în mod practic paralelismul inerent unor aplicatii prin arhitectura hardware, paralela si ea, a SMM, trebuie parcurse urmatoarele etape:

1. Identificarea paralelismului potential al aplicatiei
2. Partitionarea aplicatiei în procese (task-uri) concurente
3. Distribuirea si sincronizarea task-urilor pe procesoare

Clasificari ale SMM-urilor:

a. SMM cu resurse globale partajate (centralizate)

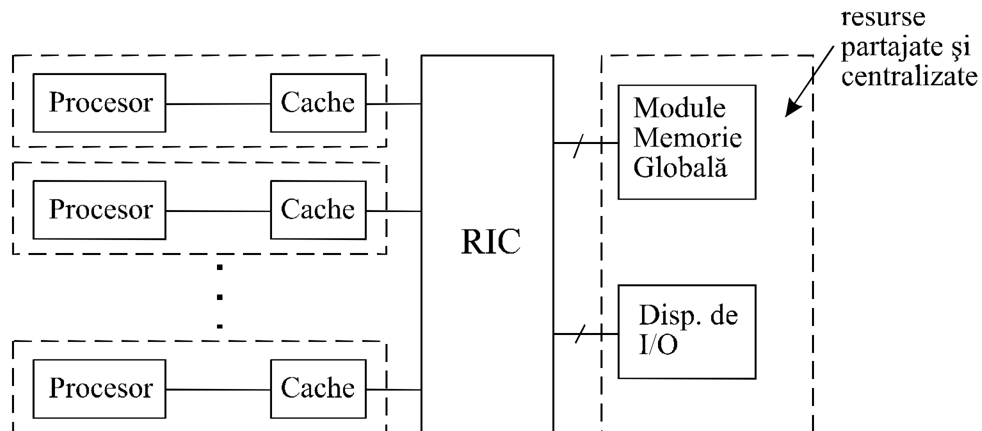


Figura 8.1. SMM cu resurse globale partajate

Procesoarele pot comunica între ele prin memoria globala (principala). Se considera ca procesoarele nu detin resurse locale, deci resursele de memorie si dispozitivele de I/O sunt comune tuturor procesoarelor urmând a fi partajate de catre acestea. Se mai numesc si sisteme UMA (Uniform Memory Access), pentru ca memoria fizica este uniform partajata de catre toate procesoarele, timpul de acces fiind în principiu acelasi pentru orice procesor. Aceste SMM, având ca RIC un bus comun, sunt cele mai populare pe plan comercial, cu precadere în aplicatiile industriale.

b. SMM cu resurse distribuite si partajate

Fiecare procesor detine în acest caz resurse locale (memorii + dispozitive de I/O). Comunicatia între 2 procesoare se poate face prin conectarea lor "punct la punct" prin intermediul RIC, ca în figura urmatoare.

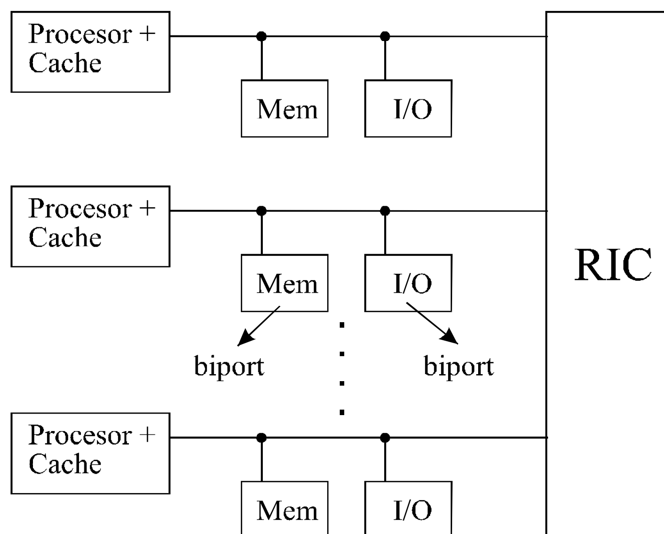


Figura 8.2. SMM cu resurse distribuite si partajate

În acest caz, resursele de memorie si porturi sunt distribuite. Aceste sisteme se mai numesc si NUMA (Non – Uniform Memory Access), în sensul ca accesarea unei anumite locatii de memorie implica o latenta mai mare sau mai mica, functie de procesorul care o acceseaza. Termenul de partajat se refera la faptul ca spatiul logic de adrese al procesoarelor este partajat, adica aceeasi adresa fizica accesata de 2 procesoare, conduce la aceeasi locatie de memorie fizica.

c. SMM hibride

Constituie o combinatie între sistemele cu resurse centralizate si partajate (caz a) si cele cu resurse distribuite si partajate (caz b). Din punct de vedere al gradului de interconectare al procesoarelor, SMM sunt de doua tipuri:

1. Sisteme strâns – cuplate (SMM propriu – zise), în care conectarea procesor – memorie se face prin intermediul unor busuri de interconectare. Sunt asociate de obicei sistemelor UMA.
2. Sisteme slab – cuplate (Retele), în care conectarea procesor – memorie se face prin legaturi seriale “punct la punct”.

8.2. ARHITECTURI CONSACRATE

1. SMM pe bus comun (RIC statica)

Caracterizata de faptul ca RIC este un simplu bus comun partajat în timp de catre μprocesoare (UMA). Este cea mai simpla arhitectura, dar conflictele potentiale ale procesoarelor (masterilor) pe busul comun, pot fi ridicate. Desigur, exista un singur master activ pe bus la un moment dat. Busul comun si memoria globala (slave) sunt partajate în timp de catre masteri. Resursele locale ale masterilor - memorii cache locale - au rolul de a diminua traficul pe busul comun. Accesul pe bus se face prin intermediul unui arbitru de prioritati, centralizat sau distribuit.

Arhitectura implica dificultati tehnologice legate de numarul maxim de masteri cuplabili (în practica pâna la 32), reflexii si diafonii ale semnalelor pe bus. Cum capacitatile si inductantele parazite cresc proportional cu lungimea busului, rezulta ca acesta trebuie sa fie relativ scurt.

Exista arhitecturi standardizate de SMM pe bus comun (VME – dezvoltat de Motorola pe μp MC680X0, MULTIBUS – Intel pentru I - 80X86).

2. SMM în inel (token – ring) – retea statica

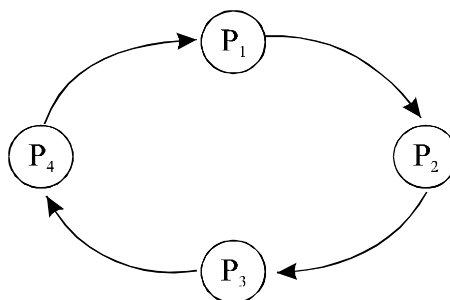


Figura 8.3. Retea Token-Ring

Arhitectura este standardizata conform standardelor IEEE 802.5. Este utilizata cu precadere în sistemele slab cuplate (retele locale). Protocolul de comunicatie are la baza trimiterea unei informatii binare speciale, numita jeton (token), de la un procesor la celalalt, în mod secvential. Un procesor P_i, nu poate sa trimita un mesaj decât daca a receptionat jetonul. Daca un

procesor doreste sa trimita un mesaj la un alt procesor, va astepta receptionarea jetonului, apoi va modifica un bit din jeton iar apoi va transmite mesajul cu jetonul modificat pe post de antet. Din acest moment, nu mai circula jeton prin structura si deci toate emisiile de mesaje sunt inhibate. Dupa ce procesorul destinatie a receptionat mesajul (date sau comenzi), îl va trimite mai departe. Conform standardului, procesorul sursa va insera din nou jetonul în structura în momentul în care si-a terminat de transmis mesajul si a receptionat "începutul" propriului mesaj emis.

Eficienta scade proportional cu numarul procesoarelor din retea.

3. SMM cu interconectare "crossbar" (retea dinamica)

Aceasta arhitectura detine complexitatea cea mai ridicata dintre arhitecturile de SMM, în schimb conflictele procesoarelor la resursele de memorie comuna partajata sunt minime. Comunicatia între orice pereche procesor – memorie este întârziata în nodul de conexiune aferent. De remarcat ca pot avea loc simultan pâna la N accese ale procesoarelor la memorie, în ipoteza în care nu exista doua procesoare care sa acceseze acelasi modul de memorie. Pentru evitarea conflictelor de acest gen, se încearca atunci când este posibil "împrastieri" favorabile ale informatiei în modulele de memorie globala. De exemplu în aplicatiile pe vectori, daca acestia au pasul 1 atunci scalarii succesivi sunt situati în blocuri succesive de memorie, rezultând minimizari ale conflictelor.

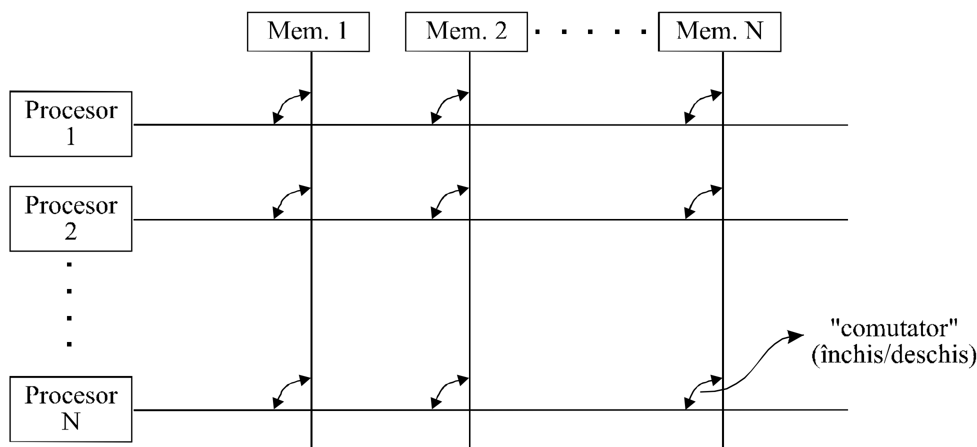


Figura 8.4. Retea crossbar

Desi cea mai performanta, arhitectura devine practic greu realizabila pentru un numar N ridicat de procesoare, din cauza costurilor ridicate (N^2 noduri).

4. SMM cu retele de interconectare dinamice multinivele

Reprezinta un compromis între SMM unibus si cele de tip crossbar. Elementul principal al RIC îl constituie comutatorul. În general sunt folosite comutatoare cu doua intrari si doua iesiri. Aceste comutatoare pot lucra "direct" sau în "cruce", adica (A→C, B→D) respectiv (A→D, B→C).

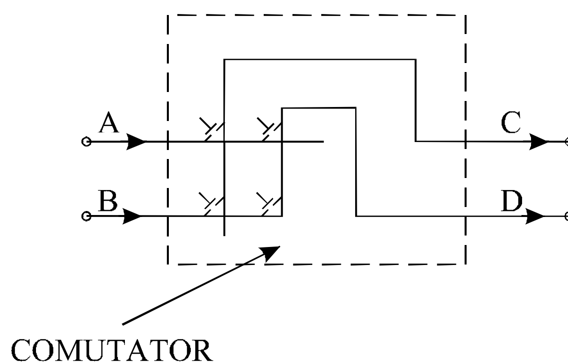


Figura 8.5. Comutator de retea

Se prezinta mai jos un SMM având o retea de interconectare pe trei nivele, într-o topologie BASELINE, cu opt procesoare si opt module de memorie (M_0, M_7).

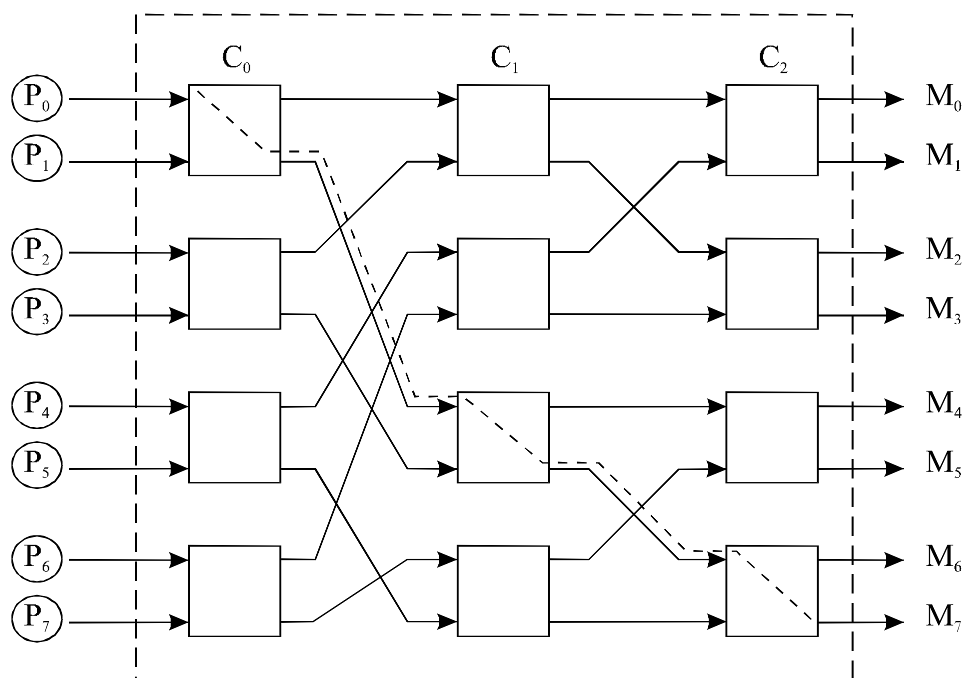


Figura 8.6. Interconectare Baseline

Cu precizarea ca $C_0 = 0$ semnifica faptul ca switch-ul C_0 lucreaza în "linie", iar $C_0 = 1$ faptul ca lucreaza în "cruce", se prezinta grafurile de comunicare "totala" procesoare – memorii pentru reteaua BASELINE.

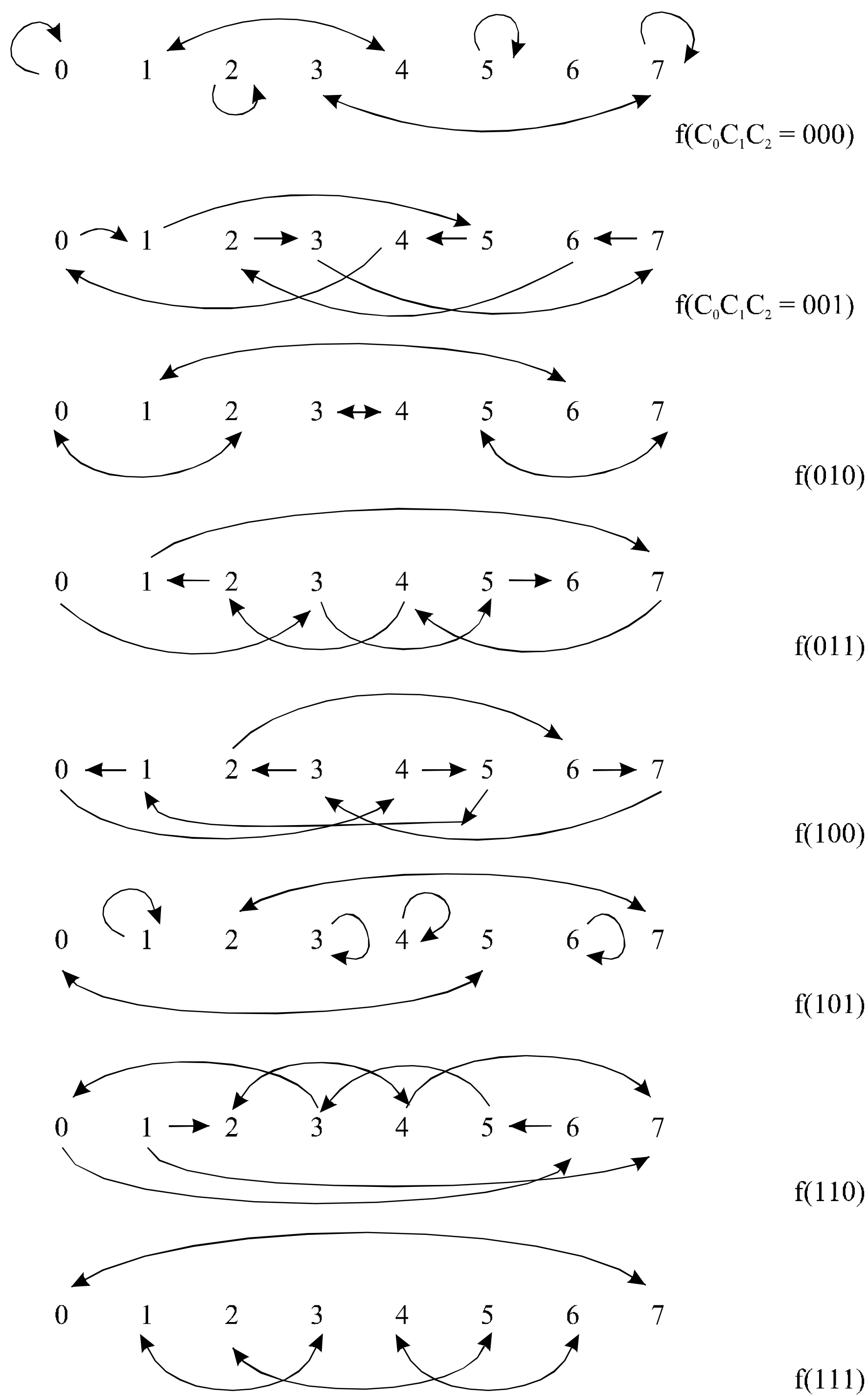


Figura 8.7. Grafuri comunicatie Baseline

Exemplu: F(100) reprezinta graful asociat retelei de interconectare procesoare – memorii, considerând ca switch-ul C_0 lucreaza în cruce iar C_1, C_2 lucreaza în linie.

Spre deosebire de arhitectura crossbar, în cazul acestor RIC – uri, nu este posibila implementarea oricarei functii bijective de comunicatie f:{P_0, P_1,..., P_7}→{M_0, M_1,..., M_7}, din cele 8! functii bijective posibile, ci doar a celor 8 functii de mai sus.

De remarcat însa ca în acest caz, complexitatea comutatoarelor este mai mare decât în cazul crossbar, în schimb sunt mai putine. Mai precis, RIC crossbar are N^2 comutatoare elementare în timp ce o astfel de retea are doar $4 \times \log_2 N \times \frac{N}{2} = 2N\log_2 N < N^2$. În schimb, o conexiune procesor-memorie este întârziata aici pe 3 nivele de comutatoare elementare si nu pe unul singur ca în cazul crossbar.

Un dezavantaj important al acestor arhitecturi, zise si "arii de procesoare" îl constituie posibilitatea unei cai de comunicare procesor – memorie de a bloca alte cai necesare. De exemplu în cazul BASELINE, calea P_0 – M_7 blocheaza interconectarea simultana a oricarora dintre urmatoarele conexiuni: P_1 – M_4, P_1 – M_5, P_1 – M_6, P_1 – M_7, P_2 – M_6, P_3 – M_6 etc. Rezulta deci ca o asemenea RIC este mai putin potrivita pentru transferuri prin comutare de circuite. În cazul unei retele cu comutare de pachete, blocarea consta în asteptarea pachetului într-un buffer asociat comutatorului, pâna când se va ivi posibilitatea trimiterii sale spre urmatorul nivel de comutatoare. Desigur, exista si alte topologii de retele multinivel (BANYAN, DELTA, etc.).

5. SMM interconectate în hipercub (statica)

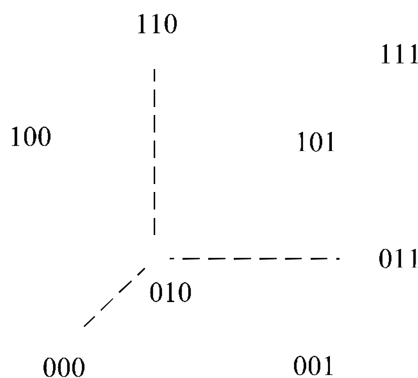


Figura 8.8. Interconectare hipercub

În hipercubul k – dimensional exista N = 2^k noduri (procesoare), fiecare de gradul k, adica având legaturi directe cu k procesoare. Daca tratam fiecare eticheta a nodurilor ca o valoare binara, nodurile conectate direct difera printr-o singura coordonata. Altfel spus, cei k vecini ai unui procesor P_j au etichete binare adiacente cu cea a lui P_j.

Pentru dimensiuni mai mari decât 3 ale lui k, diagrama de interconectare devine mai dificila dar ideea ramâne aceeasi. Câteva companii incluzând INTEL, NCUBE, FPS etc. studiaza activ masini în aceasta retea.

Totusi un SMM în sensul clasic reprezinta un sistem unibus, cu memorie centrala partajata.

8.3. GRANULARITATE SI COMUNICARE

RIC este esentiala în performanta unui SMM. Principalele criterii de performanta de care se tine cont în proiectarea unei RIC sunt:

- Întârzierea, adica timpul de transmitere pentru un singur cuvânt (mesaj)
- Largimea de banda, adica ce trafic de mesaje poate suporta reteaua în unitatea de timp
- Gradul de conectivitate, adica numarul de vecini directi pentru fiecare nod
- Costul hardware, adica ce fractie din costul total al hardului reprezinta costul RIC
- Fiabilitatea si functionalitatea (arbitrare, întreruperi).

Ideal ar fi ca un SMM dotat cu N procesoare sa proceseze un program de N ori mai rapid decât un sistem monoprocesor, cerinta numita scalabilitate completa. În realitate, acest deziderat de scalabilitate nu se realizeaza din multiple motive. În privinta scalabilitatii, aceasta este mai usor de realizat pe un sistem cu resurse distribuite decât pe unul având resurse centralizate, întrucât acestea din urma constituie un factor de “strangulare” a activitatii.

Dintre cauzele care stau în calea unei scalabilitati ideale, se amintesc:

1. Gradul de secventialitate intrinsec al algoritmului executat. Asa de exemplu, exista în cadrul unui algoritm operatii secventiale dependente de date si deci imposibil de partajat în vederea procesarii lor paralele pe mai multe procesoare.

$$\left.\begin{array}{l} n = y + z; \\ a = n + b; \end{array}\right\} \text{scalar secvential dependent de RAW}$$ 100% = secvential

----

$$\left.\begin{array}{l} \text{for } i = 1 \text{ to } 10 \\ \quad A(i) = B(i) + C(i); \end{array}\right\} \text{paralelizabil pe 10 procesoare}$$

(1-f)×100%=paralelizabil

Accelerarea S pentru un SMM cu N procesoare este, prin definitie:

$$S = \frac{T_S}{T_N}$$

unde:

T_s = timpul de executie pentru cel mai rapid algoritm secvential care rezolva problema pe un monoprocesor (SISD)

T_N = timpul de executie al algoritmului paralel executat pe un SMM cu N µprocesoare.

Daca notam cu f = fractia (procentajul) din algoritm care are un caracter eminamente secvential, $f \in [0,1]$, putem scrie:

$$T_N = f \cdot T_S + \frac{(1-f) \cdot T_S}{N},$$

adica

$$S = \frac{T_S}{f \cdot T_S + \frac{(1-f) \cdot T_S}{N}}$$

sau:

$$S = \frac{1}{f + \frac{(1-f)}{N}}$$

Legea lui G. Amdahl, $1 \le S \le N$ (scalabil)

Legea lui G. Amdahl sugereaza ca un procentaj (fx100%) oricât de scazut de calcule secventiale impune o limita superioara a accelerarii (1/f) care poate fi obtinuta pentru un anumit algoritm paralel pe un SMM, indiferent de numarul N al procesoarelor din sistem si topologia de interconectare a acestora.

1. Timpul consumat cu sincronizarea si comunicarea între procesele rezidente pe diversele (µ)procesoare din sistem.

2. Imposibilitatea balansarii optimale a activitatii procesoarelor din sistem, adica frecvent nu se poate evita situatia în care anumite procesoare sa fie practic inactive sau cu un grad scazut de utilizare.
3. Planificarea suboptimala a proceselor d.p.d.v. software (activare proces, punere în asteptare a unui proces, schimbarea contextului în comutarea proceselor etc.)
4. Operatiile de I/O, în cazul nesuprapunerii lor peste activitatea de executie a task-ului de catre procesor.

Un parametru important care influenteaza direct performanta unui SMM, e dat de granularitatea algoritmului de executat, adica dimensiunea medie (numar de instructiuni, timp de executie etc.) a unei unitati secventiale de program (USP). Prin USP se întelege o secventa de program în cadrul algoritmului paralel în cadrul careia nu se executa operatii de sincronizare sau de comunicare date cu alte procese. Se mai defineste si un alt parametru, numit timp mediu de comunicatie între 2 task-uri nerezidente pe acelasi procesor.

Din punct de vedere al raportului între granularitatea algoritmului (G) si timpul de comunicatie (C), se citeaza 2 tipuri de algoritmi:

a. Coarse grain parallelism (paralelism intrinsec masiv), caracterizati de un raport G/C relativ “mare”. Acesti algoritmi se preteaza cu bune rezultate la procesoare pe sisteme paralele de tip SIMD (vectoriale) sau MIMD (multiprocesoare).
b. Fine grain parallelism (paralelism intrinsec redus), caracterizati de un raport G/C relativ mic. În acest caz nu este recomandabila multiprocesarea din cauza granularitatii fine a algoritmului si a timpilor mari de comunicatie/ sincronizare între procesoare (USP-uri), elemente ce vor reduce dramatic accelerarea SMM. Se recomanda exploatarea paralelismului redus prin tehnici monoprocesor de tip “Instruction Level Parallelism”.

O problema majora, practic deschisa la ora actuala, o constituie scrierea de software pentru SMM. Sarcina unui compilator SMM este dificila întrucât trebuie sa determine o metoda de a executa mai multe operatii (programe), pe mai multe procesoare, în momentele de timp neprecizabile. Apoi, trebuie optimizat raportul G/C printr-o judicioasa partitionare – în general statica – a algoritmului de executat.

8.4. MODELE ANALITICE DE ESTIMARE A PERFORMANTEI

1. Un model "pesimist"

Se considera pentru început un sistem biprocesor care trebuie sa execute un program (aplicatie) ce contine M task-uri. Se presupune ca fiecare task se executa în "G" unitati relative de timp si ca oricare 2 task-uri nerezidente pe acelasi procesor consuma "C" unitati relative de timp pentru intercomunicatii (schimburi de date + sincronizari). Se va considera ca "K" task-uri se vor executa pe un procesor iar (M-K) task-uri pe celalalt, (∀) k=1,2,...,M.

Pesimismul modelului consta în faptul ca se va considera ca nu e posibila nici o suprapunere între executia unui task si comunicarea acestuia cu celelalte task-uri nerezidente pe acelasi procesor. În acest caz, timpul de executie (ET) aferent aplicatiei este dat de relatia:

$$ET = G \cdot Max(M - K, K) + C \cdot (M - K) \cdot K$$

Particularizând pentru M=50. G/C=10 (granularitate "mica"), G=1 (un task se executa într-o singura unitate de timp), se obtine:

$$ET = \underbrace{Max(50 - K, K)}_{\substack{T_R = \text{timp rulare} \\ \text{efectiva aplicatie}}} + \underbrace{0{,}1 \cdot (50 - K) \cdot K}_{\substack{T_R = \text{timp comunicatii} \\ \text{interprocesor}}}$$

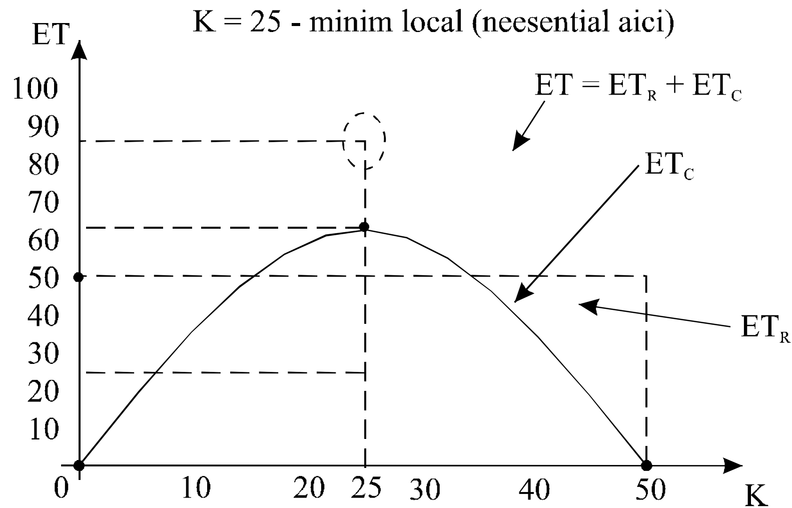


Figura 8.9. Optimizare în cazul G/C "mic"

În acest caz rezulta $k_{optim} = 0$, adica optim e ca toate cele 50 taskuri sa ruleze pe un singur procesor (monoprocesare!). Daca am considera granularitatea aplicatiei G/C = 50 ("mare"), se va obtine:

$$ET = Max(50 - K, K) + \frac{1}{40} \cdot (50 - K) \cdot K$$

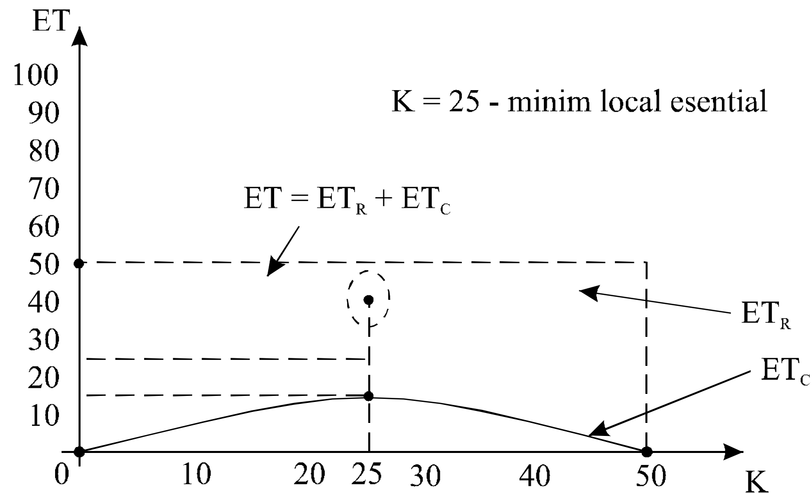


Figura 8.10. Optimizare în cazul G/C "mare"

În acest caz $k_{optim} = M/2 = 25$, adica o distribuire uniforma a numarului de task-uri pe cele 2 procesoare ("coarse grain parallelism").

În concluzie, pentru un sistem biprocesor în conditiile date, strategia optimala în vederea obtinerii performantei (ET) maxime este:

a) Daca $\frac{G}{C} \le \frac{M}{2} \Rightarrow k_{optim} = 0$ (monoprocesare)

b) Daca $\frac{G}{C} > \frac{M}{2} \Rightarrow k_{optim} = M/2$ (procesare paralela omogena)

Generalizare pentru N procesoare

Se considera pentru început un SMM cu 3 procesoare P1, P2, P3, care are de executat o aplicatie având M task-uri. Se presupune ca P1 executa k1 task-uri, P2 executa k2 task-uri iar P3 executa k3 task-uri, k1+k2+k3=M.

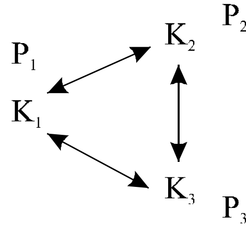


Figura 8.11. Ansambluri procesor - task

Rezulta imediat ca timpul de comunicatii (ET_c) intertask-uri va fi:

$$ET_c = \frac{C}{2}\left[(k_2 + k_3)\cdot k_1 + (k_1 + k_3)\cdot k_2 + (k_1 + k_2)\cdot k_3\right] = \frac{C}{2}\sum_{i=1}^{3} k_i (M - k_i)$$

Prin urmare, pentru un sistem cu N procesoare avem:

$$ET = G \cdot Max(k_i) + \frac{C}{2}\sum_{i=1}^{N} k_i (M - k_i) \text{ sau:}$$

$$ET = G \cdot Max(k_i) + \frac{C}{2}\left(M^2 - \sum_{i=1}^{N} k_i^2\right), \quad (\forall)\, i = \overline{1, N}$$

În ipoteza - nu neaparat optimala - unei alocari uniforme a task-urilor pentru toate cele N procesoare, avem: $\left(k_i = \frac{M}{N}\right)$

$$ET_N = \frac{G \cdot M}{N} + \frac{C \cdot M^2}{2} - \frac{C \cdot M^2}{2N}$$

Timpul de executie al algoritmului pe un singur procesor este:

$$ET_1 = G \times M$$

"Punctul critic" se atinge când $ET_N = ET_1$, adica:

$$\frac{G \cdot M}{N} + \frac{C \cdot M^2}{2} = \frac{C \cdot M^2}{2N} + G \cdot M \text{ , sau:}$$

$$G\left(1 - \frac{1}{N}\right) = \frac{C \cdot M}{2}\left(1 - \frac{1}{N}\right)\text{, adica:}$$

$\frac{G}{C} = \frac{M}{2}$ (conditie de performanta identica mono-multi)

În concluzie:

a) Daca $\frac{G}{C} \geq \frac{M}{2}$, este indicata multiprocesare omogena (coarse grain)

b) Daca $\frac{G}{C} < \frac{M}{2}$, monoprocesarea e mai indicata (fine grain)

2. Un model mai "optimist"

Sa presupunem acum ca executia task-urilor se poate suprapune cu comunicatia interprocesoare. În acest caz avem:

$$ET = Max\left\{G \cdot Max(k_i), \frac{C}{2}\sum_{i=1}^{N} k_i \cdot (M - k_i)\right\}$$

Optimul s-ar obtine atunci când suprapunerea celor 2 componente ar fi perfecta, adica:

$$\frac{G \cdot M}{N} = \frac{C \cdot M^2}{2} - \frac{C \cdot M^2}{2N^2} \Rightarrow \frac{G}{N} = \frac{C \cdot M}{2}\left(1 - \frac{1}{N}\right)$$

Pentru un numar "mare" de procesoare (N) avem:

$$\frac{G}{N} \cong \frac{C \cdot M}{2} \Rightarrow N_{optim} = \frac{2}{N} \cdot \frac{G}{C}$$

Obs. 1. N creste liniar cu granularitatea (G/C) aplicatiei

Obs. 2. Aparent paradoxal, N_{optim} este invers proportional cu numarul de task-uri paralele M în care s-a împartit aplicatia. Nu este nici un paradox pentru ca cresterea lui M, determina cresterea timpului de comunicatii interprocesor (ET_c).

3. Un model cu costuri liniare de comunicatie

Sa consideram acum un model de SMM în care costurile de comunicatie sa fie liniare cu numarul de procesoare N si nu cu numarul de task-uri asigurate fiecarui procesor (P_i), ca în modelul precedent.

În acest caz, putem scrie:

$$ET_N = G \cdot Max(k_i) + C \cdot N$$

Sa determinam în continuare, pâna la ce "N", avem ca $ET_N > ET_{N+1}$, adica sa determinam în acest caz un numar optim de procesoare (N_{opt}) care sa ruleze cele M task-uri.

Pentru simplificare, se considera o distributie uniforma de task-uri/procesor $\left(k_i = \frac{M}{N}\right)$, adica:

$$ET_N = \frac{G \cdot M}{N} + C \cdot N$$

$$\Rightarrow \Delta_N = ET_N - ET_{N+1} = G \cdot M\left(\frac{1}{N} - \frac{1}{N+1}\right) - C \text{, adica:}$$

$$\Delta_N = \frac{G \cdot M}{N(N+1)} - C \Leftrightarrow \frac{G}{C} = \frac{N(N+1)}{m}$$

Rezulta deci $N_{optim} \cong \sqrt{\frac{G}{C} \cdot M}$.

Asadar N_{optim} nu creste proportional cu numarul de task-uri M ci proportional cu $\sqrt{M}$ si de asemenea, proportional cu radacina patrata a

granularitatii $\sqrt{\frac{G}{C}}$. Pentru un N > N_{optim}, performanta SMM cu costuri liniare de comunicatie, se degradeaza datorita costurilor comunicatiilor interprocesor.

Obs. Toate aceste modele sunt simpliste, generând rezultate contradictorii chiar si discutabile. Complexitatea procesarii SMM e prea mare pentru abordari analitice de referinta. Se impun si aici simularile pe benchmark-uri specifice.

8.5. ARHITECTURA SISTEMULUI DE MEMORIE

8.5.1. DEFINIREA PROBLEMEI

Într-un SMM o informatie poate fi privata (locala) - daca ea este utilizata de catre un singur CPU, sau partajata - daca se impune a fi utilizata de catre mai multe procesoare. În cazul informatiilor partajate, de obicei acestea sunt "casate" (memorate în cache-urile locale ale unui anumit CPU), reducându-se astfel latenta necesara accesarii lor de catre un anumit procesor. Totodata, prin aceasta casare a informatiilor partajate se reduc coliziunile implicate de accesul simultan (citiri) al mai multor CPU-uri la respectiva informatie si respectiv necesitatile de largime de banda a busului comun.

Din pacate, pe lânga aceste avantaje, casarea introduce un dezavantaj major, constând în problema coerentei cache-urilor. În principiu, un sistem de memorie aferent unui SMM este coerent daca orice citire a unei informatii returneaza informatia scrisa cel mai recent. Cu alte cuvinte, problema coerentei cache-urilor în SMM se refera la necesitatea ca un bloc din memoria globala - memorat în mai multe memorii cache locale - sa fie actualizat în toate aceste memorii, la orice acces de scriere al unui anumit CPU.

D.p.d.v. al acceselor de scriere a unui procesor, exista 2 posibilitati:

1) Strategia *"Write - Through"* (WT), prin care informatia este scrisa de catre CPU atât în blocul aferent din cache cât si în blocul corespunzator din

memoria globala (în acest caz, pentru a reduce stagnarea µprocesorului datorita accesului mai lent la memoria globala, de multe ori sarcina de scriere în memoria globala e alocata unui procesor de iesire specializat, permitându-se astfel CPU-ului sa-si continue programul fara stagnari).

2) Strategia *"Write - Back"* (WB), prin care informatia este scrisa numai în cache-ul local. Blocul modificat din acest cache, va fi scris în memoria globala numai când acesta va fi evacuat din cache. În cazul acestei strategii, pentru a reduce frecventa scrierilor în memoria globala, exista asociat fiecarui bloc din cache un asa-zis "dirty bit" (bit D), care indica daca blocul din cache a fost sau nu a fost modificat. Daca nu a fost, atunci nu mai are sens evacuarea sa efectiva în memoria globala iar scrierile se fac relativ la timpul de acces al cache-ului. De asemenea, multiplele scrieri într-un bloc din cache necesita o singura scriere (evacuarea) în memoria globala.

În cadrul SMM tehnica WB e atractiva pentru ca se reduce traficul pe busul comun, în timp ce tehnica WT este mai usor de implementat si ar putea gestiona poate mai facil coerenta cache-urilor.

Revenind acum la problema coerentei cache-urilor într-un sistem de memorie centralizat si partajat, se va considera un exemplu care arata cum 2 procesoare pot "vedea" 2 valori diferite pentru aceeasi locatie (X) de memorie globala, adica un caz tipic de incoerenta a unei valori globale.

Pas	Eveniment	Continut cache CPU1	Continut cache CPU2	Continut Memorie globala (X)
0	—	?	?	1
1	CPU1 citeste X	1	?	1
2	CPU2 citeste X	1	1	1
3	CPU1 scrie 0 în X (WT)	0	1	0
	WB	0	1	1

Tabelul 8.1.

Exemplificarea unei incoerente

S-a presupus ca initial, nici una din cele 2 cache-uri nu contine variabila globala X si ca aceasta are valoare 1 în memoria globala. De asemenea s-au presupus cache-uri de tip WT (un cache WB ar introduce o incoerenta asemanatoare). În pasul 3 CPU 2 are o valoare incoerenta a variabilei X.

Definitie:

Un SMM este coerent daca sunt îndeplinite urmatoarele 3 conditii:

1. Un procesor P scrie variabila X. Daca dupa un timp, P va citi variabila X si daca între cele 2 accese la memorie ale lui P nici un alt procesor nu a scris în X, atunci P va citi aceeasi valoare a lui X cu cea scrisa. Afirmatia nu este chiar triviala având în vedere cache-urile locale (evacuari).
2. Un procesor Pi scrie variabila X. Daca dupa un timp, Pj va citi variabila X si daca între timp nici un alt procesor nu a scris în X, atunci Pj va citi aceeasi valoare ca cea scrisa de catre Pi. Conditia nu e triviala, având în vedere exemplul de mai sus (CPU1 scrie "0", CPU2 citeste "1").
3. Scrierile la aceeasi locatie (X) trebuie serializate prin arbitrare. De exemplu daca P1 scrie 1 la X si apoi P2 scrie 2 la X, niciodata un procesor nu va putea citi întâi X=2 si apoi X=1.

O nerespectare a unuia din cele 3 principii, conduce la incoerenta sistemului de memorie.

8.5.2. PROTOCOALE DE ASIGURARE A COERENTEI CACHE-URILOR

Fiecare dintre cache-urile care contine copia unui bloc din memoria globala, contine de asemenea "starea" acelui bloc d.p.d.v. al procesului de partajare (partajat - "read-only", exclusiv - "read/write", invalid). Asadar, nu exista o centralizare a informatiei de stare a blocului. Fiecare controller de cache, monitorizeaza permanent busul comun pentru a determina daca cache-ul respectiv contine sau nu contine o copie a blocului cerut pe busul comun (*snooping protocol*). În cadrul acestui protocol de monitorizare , exista 2 posibilitati de mentinere a coerentei functie de ceea ce se întâmpla la o scriere:

1) *Write Invalidate* (WI). Procesorul care scrie determina ca toate copiile din celelalte memorii cache sa fie invalidate (se pune bitul de validare V=0 în cadrul blocului respectiv din cache ⇒ orice acces la acel bloc va fi cu MISS), înainte însa ca el sa-si modifice blocul în cache-ul propriu. Respectivul procesor va activa pe busul comun un semnal de invalidare bloc si toate celelalte procesoare vor verifica prin monitorizare daca detin o copie a blocului; daca DA, trebuie sa invalideze blocul care contine acel cuvânt. Evident ca un anumit bloc invalidat în cache, nu va mai fi scris în memoria globala la evacuare. Astfel, WI permite mai multe citiri simultane a

blocului dar o singura scriere în bloc la un anumit moment dat. Este foarte des implementat în SMM.

2) *Write Broadcast (Write Update* - WBC). Procesorul care scrie pune data de scris pe busul comun spre a fi actualizate toate copiile din celelalte cache-uri. Pentru asta, este util sa se stie daca un cuvânt este sau nu este partajat (continut în alte cache-uri decât cel al procesorului care scrie). Daca nu e partajat într-un anumit cache, evident ca actualizarea (*updating*) sa în acel cache este inutila. Este mai putin utilizat în SMM.

Obs. Ambele strategii de mentinere a coerentei (WI, WBC) pot fi asociate cu oricare dintre protocoalele de scriere în SMM (WT respectiv WB).

În continuare, se prezinta un exemplu de protocol de coerenta WI, bazat pe un protocol de scriere în cache de tip WB.

Pas	Activitate procesor	Activitate pe bus comun	Loc.X cache CPU1	Loc.X cache CPU2	Loc. X Memorie globala
0			?	?	0
1	CPU1 citeste X	Cache Miss (X)	0	?	1
2	CPU2 citeste X	Cache Miss (X)	0	0	0
3	CPU1 scrie ‘1’ în X	Invalidare X	1	0 INV.	0
4	CPU2 citeste X	Cache Miss (X)	1	1	1

Tabelul 8.2.

Coerenta prin protocol WI

În pasul 4, CPU1 aborteaza ciclul de citire al lui CPU2 din memoria globala si pune pe busul comun valoarea lui X ("1", copie exclusiva). Apoi, scrie (actualizeaza) valoarea lui X în cache-ul lui CPU2 si în memoria globala iar X devine o variabila partajata.

Mai jos, se prezinta un exemplu de protocol de coerenta WBC, bazat pe un protocol de scriere în cache de tip "Write Through":

Pas	Activitate procesor	Activitate pe bus comun	Loc.X cache CPU1	Loc.X cache CPU2	Loc. X Memorie globala
0			—	—	0
1	CPU1 citeste X	Cache Miss (X)	0	—	0
2	CPU2 citeste X	Cache Miss (X)	0	0	0
3	CPU1 scrie ‘1’ în X	Write update X	1	1	1
4	CPU2 citeste X (HIT)	—	1	1	1

Tabelul 8.3.

Coerenta prin protocol WBC

Diferentele de performanta între protocoalele de coerenta WI si WBC, provin în principal din urmatoarele caracteristici:

a) Daca acelasi CPU scrie de mai multe ori la aceeasi adresa fara aparitia intercalata a unor citiri din partea altor procesoare sunt necesare scrieri multiple pe busul comun în cazul WBC, în schimb e necesara doar o invalidare initiala în cazul WI.
b) WBC lucreaza pe cuvinte (deci la scrieri repetate în acelasi bloc acceseaza repetat busul comun), în timp ce WI lucreaza pe bloc (la scrieri repetate într-un bloc, determina doar o invalidare initiala a acelui bloc din celelalte cache-uri care îl contin).
c) Întârzierea între scrierea unui cuvânt de catre un CPU si citirea respectivei valori de catre un alt procesor, este în general mai mica într-un protocol WBC (hit) decât într-unul WI (miss).
d) Strategia WI, prin invalidarea blocurilor din celelalte cache-uri, mareste rata de miss. În schimb, strategia WBC mareste traficul pe busul comun.

Actualmente, strategia WI este preferata în majoritatea implementarilor. În continuare, se va considera un protocol de coerenta WI si unul de scriere de tip WB. Pentru implementarea protocolului WI, procesorul acceseaza busul comun si distribuie pe acest bus, adresa de acces spre a fi invalidata în toate copiile partajate. Toate procesoarele, monitorizeaza continuu busul, în acest scop. Daca adresa respectiva este casata, atunci data aferenta este invalidata.

În plus fata de procesele de invalidare, este de asemenea necesar în cazul unui miss în cache, sa se localizeze data necesara. În cazul WT e simplu, cea mai recenta copie a blocului respectiv se afla în memoria globala. Pentru un cache "Write Back" însa, problema e mai complicata întrucât cea mai recenta valoare a datei respective se afla într-un cache mai

degraba decât în memoria globala. Solutia consta în urmatoarea: daca un anumit procesor detine o copie a blocului accesat, având bitii D=1 (dirty) si V=1 (valid), atunci el furnizeaza pe busul comun data respectiva si aborteaza accesul la memoria globala (vezi pasul 4, tabel sus, pagina anterioara).

În cazul unei scrieri cu MISS (WI, WB), invalidarea blocului în care se scrie, nu are sens sa se faca daca nici un alt cache nu îl contine. Rezulta ca e necesar sa se stie starea unui anumit bloc (partajat/nepartajat) în orice moment. Asadar în plus fata de bitii de stare D si V, fiecare bloc mai detine un bit P (partajat sau nu). O scriere într-un bloc partajat (P=1), determina invalidarea pe bus si marcheaza blocul ca "privat" (P=0), adica respectivul cache detine copia exclusiva a acelui bloc. Procesorul respectiv se numeste în acest caz, proprietar (owner). Daca ulterior, un alt procesor citeste blocul respectiv, acesta devine din nou partajat (P=1).

Un protocol de coerenta (WI,WB) se implementeaza printr-un microcontroller de tip automat finit, dedicat, plasat în fiecare nod. Controllerul raspunde atât la cererile CPU-ului propriu cât si la cererile de pe busul comun. Ca urmare a acestor cereri, controllerul modifica "starea" blocului din cache-ul local si utilizeaza busul pentru accesarea sau invalidarea informatiei. Un bloc din cache poate fi într-una din urmatoarele 3 stari: invalid, partajat (read only) si exclusiv (read - write). Pentru simplitate, protocolul implementat nu va distinge între un "write hit" si un "write miss" la un bloc "partajat" din cache; ambele se vor trata ca "write miss"-uri. Când pe busul comun are loc un "write miss", fiecare CPU care contine în cache o copie a blocului accesat pe bus, o va invalida. Daca blocul respectiv este "exclusiv" (doar în acel cache), se va actualiza ("write back"). Orice tranzitie în starea "exclusiv" (la o scriere în bloc), necesita aparitia unui "write miss" pe busul comun, cauzând invalidarea tuturor copiilor blocului respectiv, aflate în alte cache-uri. Aparitia unui "read miss" pe bus, la un bloc exclusiv, va determina punerea lui ca "partajat" de catre procesorul proprietar.

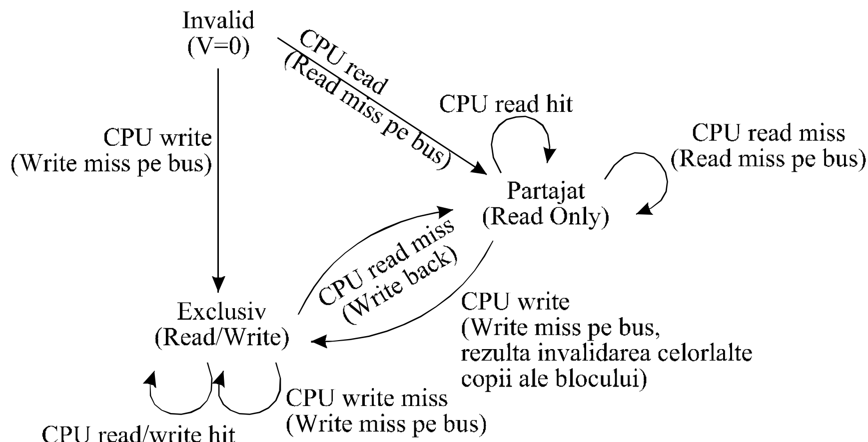


Figura 8. 12. Tranzitiile "starii" blocului bazat pe cererile CPU

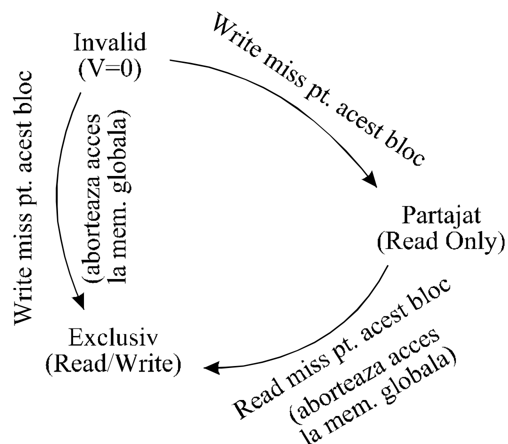


Figura 8. 13. Tranzitiile "starii" blocului bazat pe cererile de pe bus

8.6. SINCRONIZAREA PROCESELOR

Cheia sincronizarii proceselor în SMM este data de implementarea unor asa-zise procese atomice. Un proces atomic reprezinta un proces care odata initiat, nu mai poate fi întrerupt de catre un alt proces. Spre exemplu, sa consideram ca pe un SMM se executa o aplicatie care calculeaza o suma globala prin niste sume locale calculate pe fiecare procesor, ca mai jos:

```
        LocSum=0;
        For i=1 to Max
                LocSum=LocSum+LocTable[i]; secventa executata în paralel de
                                           ; catre fiecare procesor!
Proces  | LOCK
Atomic  |         GlobSum=GlobSum+LocSum;
        | UNLOCK
```

Procesul LOCK/UNLOCK este atomic, în sensul ca numai un anumit procesor executa acest proces la un moment dat. În caz contrar s-ar putea obtine rezultate hazardate pentru variabila globala "GlobSum". Astfel de exemplu, P1 citeste GlobSum = X, P2 la fel, apoi P1 scrie GlobSum = GlobSum + LocSum1 iar apoi P2 va scrie GlobSum = GlobSum + LocSum2 = X + LocSum2 (incorect !). Este deci necesar ca structura hardware sa poata asigura atomizarea unui proces software.

O solutie consta în implementarea în cadrul µprocesoarelor actuale a unor instructiuni atomice de tip "Read - Modify - Write", neîntreruptibile la nivelul unei cereri de bus (BUSREQ). Cu precizarea ca o variabila globala rezidenta în memoria comuna ("GlobSum") detine un octet "Semafor" asociat, care indica daca aceasta este sau nu este ocupata, se prezinta un exemplu de implementare a procesului atomic precedent (LOCK UNLOCK), pe un SMM cu µprocesoare INTEL - 8086:

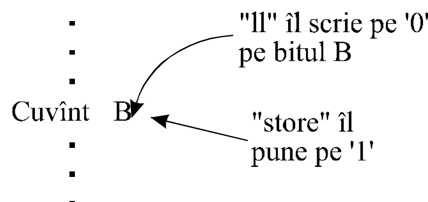
```
      MOV AL,01
WAIT:LOCK XCHG AL, <semafor>;  instructiune atomica "Read -
                              ; Modify - Write"
      TEST 01,AL              ; resursa “GlobSum” este libera?
      JNZ WAIT                ; daca nu, repeta
      MOV AX, <GlobSum>       ; AX ← (GlobSum)
      ADD AX,BX               ; GlobSum = (GlobSum)+(LocSum)
      MOV <GlobSum>, AX       ; AX → GlobSum
      XOR AL,AL; AL ≡ 0
      MOV <semafor>, AL       ; eliberare resursa
```

Obs. S-a presupus ca fiecare proces local a depus rezultatul "LocSum" în registrul BX al fiecarui procesor.

O implementare diferita a acestor procese atomice este realizata în µprocesoarele mai recente. Se porneste de la constatarea faptului ca instructiunile atomice tip "Read - Modify - Write" sunt dificil de

implementat, necesitând ajutorul hardware - ului. O alternativa o constituie perechea de instructiuni "load locked" ("ll") si "store-ul conditionat" ("sc"). Aceste instructiuni sunt utilizate în secventa: daca continutul locatiei de memorie specificata de "ll" se modifica înainte de aparitia unui "sc" la aceeasi locatie de memorie, atunci acesta ("sc") nu se executa propriu-zis. La fel, daca procesorul proceseaza o comutare de context (ex. CALL/RET, întreruperi etc.) între cele 2 instructiuni, de asemenea "sc"-ul practic nu se face. Practic instructiunea conditionata "sc Reg, Adresa", returneaza în registrul "reg" '1' sau '0' dupa cum s-a facut sau nu s-a facut. Cu aceste precizari, se prezinta implementarea unei operatii atomice de tipul "$(R_4) \leftrightarrow Mem|_{adr(R1)}$":

```
rep:    mov R3,R4
        ll R2, 0(R1)
        sc R3, 0(R1)
        begz R3,rep
        mov R4,R2
```



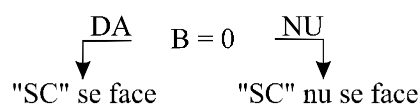


Figura 8. 14. Implementarea unei operatii atomice

O alta primitiva utila de sincronizare este cea de tip "fetch and increment". Ea returneaza valoarea unei locatii de memorie si o incrementeaza atomic. Iata mai jos un "fetch and increment" atomic, implementat prin mecanismul "ll/sc":

```
rep:    ll R2, 0(R1)
        add R2,R2,#1
        sc R2, 0(R1)
        begz R2,rep
```

8.6.1. ATOMIZARI SI SINCRONIZARI

În cazul proceselor de sincronizare, daca un anumit procesor "vede" semaforul asociat unei variabile globale pe '1' (LOCK - ocupat), are 2 posibilitati de principiu:

a) Sa ramâna în bucla de testare a semaforului pâna când acesta devine '0' (UNLOCK) - strategia "spin lock"
b) Sa abandoneze intrarea în respectivul proces, care va fi pus într-o stare de asteptare si sa initieze un alt proces disponibil - comutare de task-uri.

Strategia a) desi des utilizata, poate prelungi mult alocarea unui proces de catre un anumit procesor. Pentru a dealoca un proces (variabila aferenta), procesorul respectiv trebuie sa scrie pe '0' semaforul asociat. Este posibil ca aceasta dealocare sa fie întârziata - dulce ironie! - datorita faptului ca simultan, alte N procesoare doresc sa testeze semaforul în vederea alocarii resursei (prin bucle de tip Read - Modify - Write). Pentru evitarea acestei deficiente este necesar ca o cerere de bus de tip "Write" sa se cableze ca fiind mai prioritara decât o cerere de bus în vederea unei operatii tip "Read - Modify - Write". Altfel spus, dealocarea unei resurse este implementata ca fiind mai prioritara decât testarea semaforului în vederea alocarii resursei. Strategia b) prezinta deficiente legate în special de timpii mari determinati de dealocarea/alocarea proceselor (salvari/restaurari de contexte).

În ipoteza ca în SMM nu exista mecanisme de mentinere a coerentei cache-urilor, cea mai simpla implemetare a verificarii disponibilitatii unei variabile globale este urmatoarea (spin lock):

```
        li R2, #1; R2 ← ‘1’
test:   lock exchg R2,0(R1)        ; atomica
        bnez R2,test
```

Daca însa ar exista mecanisme de coerenta a cache-urilor, semaforul ar putea fi atasat local. Primul avantaj ar consta în faptul ca testarea semaforului ('0' sau '1') s-ar face din cache-ul local fara sa mai fie necesara accesarea busului comun. Al 2-lea avantaj - de natura statistica - se bazeaza pe faptul dovedit, ca e probabil ca într-un viitor apropiat procesorul respectiv sa doreasca sa testeze din nou semaforul (localitate spatiala si temporala).

În vederea obtinerii primului avantaj, bucla anterioara trebuie modificata. Fiecare "exchg" implica o operatie (ciclu) de tip "Write". Cum secventa anterioara de testare a semaforului poate fi executata în paralel de

mai multe procesoare, se pot genera concurential mai multi cicli (cereri) de tip "Write". Cele mai multe, vor conduce la miss-uri, întrucât fiecare procesor încearca sa obtina semaforul într-o stare "exclusiva".

Asadar, bucla se va modifica în acest caz ca mai jos:

```
test:   lw R2,0(R1)    ;        testare pe copia locala
        bnez R2, test           ; a semaforului
        li R2, #1               ; setare concurentiala a
        lock exchg R2,0(R1)     ; semaforului de catre
        bnez R2,test            ; procesoare (un singur câstigator)
```

Sa studiem acum implicatiile acestei secvente de program într-un SMM cu 3 procesoare P0, P1, P2 implementând un protocol de coerenta a cache-urilor de tip WI si un protocol de scriere în cache de tip "Write Back".

Pas	Procesor P_0	Procesor P_1	Procesor P_2	Stare semafor	Activitate pe BUS-ul comun
1	Are Sem = 1 (LOCK) pus chiar de el	Testare Sem=0? NU!	Testare Sem=0? NU!	Partajat	-
2	Termina proces si pune Sem = 0 în cache	Receptioneaza invalidare în cache	Receptioneaz a invalidare în cache	Exclusiv	Write invalidate pentru "Sem" de la P_0
3	-	Read miss	Read miss	Partajat	Arbitru serveste pe P_2; Write back de la P_0
4	-	WAIT (acces la bus)	Sem = 0	Partajat	Read miss-ul pentru P_2 satisfacut
5	-	Sem = 0	Executa "exchg" ⇒ cache miss	Partajat	Read miss-ul pentru P_1 satisfacut
6	-	Executa "exchg" ⇒ cache miss	Terminare "exchg". Primeste '0', scrie Sem = 1	Exclusiv	P_2 servit; Write invalidate "Sem"
7	-	Terminare "exchg". Primeste '1' ⇒ LOCK!	Intra în sectiunea critica" de program	Partajat	P_1 servit

8	-	Testeaza în prima bucla daca "Sem" = 0?	-	-	-

Tabelul 8.4.

Conlucrarea a trei procese într-un SMM

Pentru a minimiza traficul pe busul comun introdus de catre instructiunea "exchg", secventa anterioara se poate rafina ca mai jos:

```
test:  ll    R2,O(R1)
       bnez R2,test
       li    R2,#1
       sc    R2,O(R1)         ;un singur Pk o va executa cu
       begz R2,test           ; succes, restul, nu ⇒ scade traficul pe bus
```

Sincronizarea la bariera

Este o tehnica de sincronizare deosebit de utilizata în programele cu bucle paralele. O bariera forteaza toate procesele sa astepte pâna când toate au atins bariera, abia apoi permitându-se continuarea acestor procese. O implementare tipica a unei bariere poate fi realizata prin 2 bucle succesive: una atomica în vederea incrementarii unui contor sincron cu fiecare proces care ajunge la bariera iar cealalta în vederea mentinerii în asteptare a proceselor pâna când aceasta îndeplineste o anumita conditie (test); se va folosi functia "spin (cond)" pentru a indica acest fapt.

Se prezinta implementarea tipica a unei bariere:

```
         LOCK(counterlock)
Proces   if(count==0) release=0        ; /*sterge release la început*/
atomic          count=count+1          ; /*contorizeazã procesul ajuns
                                        la bariera*/
                UNLOCK(counterlogic)
                if(count==total)
                       { /*toate procesoarele ajunse!*/
                       count=0;
                       release=1;
                }
                else
                { /*mai sunt procese de ajuns*/
```

```
                    spin(release=1);   /*asteapta   pâna   ce   ajunge   si
   ultimul*/
                    }
```

"total" – nr. maxim al proceselor ce trebuie sa atinga bariera
"release" – utilizat pentru mentinerea în asteptare a proceselor la bariera

Exista totusi posibilitatea de exemplu, ca un procesor (proces) sa paraseasca bariera înaintea celorlalte care ar sta în bucla "spin (release=1)" si ar rezulta o comutare de task-uri chiar în acest moment. La revenire vor vedea "release=0" pentru ca procesul care a iesit a intrat din nou în bariera. Rezulta deci o blocare nedorita a proceselor în testarea "spin".

Solutia în vederea eliminarii acestui hazard consta în utilizarea unei variabile private asociate procesului (local_sense). Bariera devine:

```
local_sense=!local_sense;
LOCK(counterlock);
count++;
UNLOCK(counterlock);
if(count==total)
{
        count=0;
        release=local_sense;
}
else
{
        spin(release=local_sense);
}
```

Daca un proces iese din bariera urmând ca mai apoi sa intre într-o noua instanta a barierei, în timp ce celelalte procese sunt înca în bariera (prima instanta), acesta nu va bloca celelalte procese întrucât el nu reseteaza variabila "release" ca în implementarea anterioara a barierei.

Obs. D.p.d.v. al programatorului secvential bariera de la pag. anterioara este corecta.

Aplicatie

Sa se însumeze elementele unui tablou A[i] continând 128.000 scalari. Procesarea se va efectua pe un SMM având 16 μprocesoare conectate la un bus comun.

Solutie:

Notam Pn numarul μprocesorului curent, Pn∈{0,1,...,15}. Programul urmator, va fi executat de catre toate μprocesoarele în paralel, implementând conceptul de *"Single Program Multiple Data"* (SPMD):

```
Sum[Pn]=0;
For(i=800*Pn;i<8000*(Pn+1);i=i+1)
        Sum[Pn]=Sum[Pn]+A[i];
        limit=16;
        half=16;
        repeat
                synch()         ; primitiva de sincronizare la bariera
                half=half/2;
                if(Pn<half) Sum[Pn]=Sum[Pn]+Sum[limit-Pn-1];
                        limit=half;
        until (half==1);
```

Obs. 1. Necesitatea sincronizarii la bariera pe parcursul unei iteratii a tuturor proceselor. Primitiva synch() se poate implementa atât prin hard cât si prin soft.

Obs. 2. Timpul de procesare este de ordinul $O[\log_2 N]$ în loc de O[N] cât ar fi luat unui sistem conventional (SISD), N = numarul de iteratii necesare.

Obs. 3. Suma totala se obtine în variabila Sum[0]. Sum[Pn], half, limit sunt variabile partajate.

Obs. 4. $8000 = \frac{128.000 < \text{scalari} >}{16 < \text{procesoare} >}$

8.7. CONSISTENTA VARIABILELOR PARTAJATE

Coerenta cache-urilor asigura o viziune consistenta a memoriei pentru diversele procesoare din sistem. Nu se raspunde însa la întrebarea "CÂT de consistenta?", adica în ce moment trebuie sa vada un procesor ca o anumita variabila a fost modificata de un altul?

Exemplu:

(P1)	(P2)
A=0;	B=0;
- - - -	- - - -
A=1;	B=1;
L1: if(B==0)...	L2: if(A==0)...

În mod normal, este imposibil pentru ambele programe rezidente pe procesoare diferite (P1, P2) sa evalueze pe adevarat conditiile L1 si L2 întrucât daca B=0 atunci A=1 si daca A=0 atunci B=1. Si totusi, acest fapt s-ar putea întâmpla. Sa presupunem ca variabilele partajate A si B sunt casate pe '0' în ambele procesoare. Daca de exemplu, între scrierea A=1 în P1 si invalidarea lui A în P2 se scurge un timp suficient de îndelungat, atunci este posibil ca P2 ajuns la eticheta L2 sa evalueze (eronat) A=0.

Cea mai simpla solutie consta în fortarea fiecarui procesor care scrie o variabila partajata, de a-si întârzia aceasta scriere pâna în momentul în care toate invalidarile (WI) cauzate de catre procesul de scriere, se vor fi terminat. Aceasta strategie simpla se numeste consistenta secventiala.

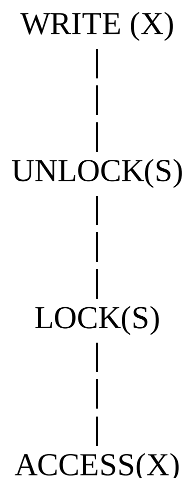
Obs. Consistenta secventiala, nu permite de exemplu implementarea unui "write buffer" la scriere, care pe baza captarii adresei si datei procesorului, sa se ocupe în continuare de procesul de scriere efectiva a datei în memorie, degrevând astfel procesorul de acest proces.

Desi consistenta secventiala prezinta o paradigma simpla d.p.d.v. al programatorului, totusi ea reduce performanta SMM, în special pe sistemele având un numar mare de procesoare sau, cu cai de interconectare de latente ridicate.

Un model de asemenea simplu d.p.d.v. al programatorului dar care permite o implementare mai eficienta, se bazeaza pe asumarea faptului ca programele aflate în executie pe diversele procesoare din sistem, sunt sincronizate. Un program este sincronizat daca toate accesele la o variabila partajata sunt ordonate prin operatii de sincronizare. Astfel, accesarea unei date este ordonata printr-o operatie de sincronizare daca, în orice instanta posibila de executie, scrierea unei variabile partajate de catre un procesor si respectiv accesarea (scriere/citire) acelei variabile de catre un alt procesor, sunt separate între ele printr-o pereche de operatii de sincronizare. O astfel de operatie este executata dupa "WRITE" de catre procesorul care scrie iar cealalta operatie este executata de catre celalalt procesor înainte de accesul sau la data partajata. Numim aceste operatii de sincronizare UNLOCK -

pentru ca deblocheaza un procesor (proces de scriere) blocat, respectiv LOCK - pentru ca îi da dreptul unui procesor de a citi o variabila partajata.

Asadar, un program este sincronizat permitând consistenta variabilelor partajate, daca fiecare "WRITE" executat de catre un procesor urmat de un acces la aceeasi data a unui alt procesor, se separa ca mai jos:



Obs. În acest caz, gestiunea consistentei este lasata în seama programatorului (S.O.)

8.8. METODE DE INTERCONECTARE LA MAGISTRALE

În SMM o functie de baza o reprezinta comunicarea între µsistemele componente pentru schimburi de date si sincronizare în vederea executiei programelor. Conceptele fundamentale referitoare la arhitectura si modul de proiectare al SMM, se împart în 2 structuri de baza:

- structuri SMM cu module functionale distribuite
- structuri SMM cu module functionale concentrate

Un modul de procesare dintr-un SMM detine 2 componente:

a) Single Board Computer-ul (SBC), compus din CPU, memorie, I/O

b) Bloc interfata la magistrala comuna a SMM (BIN)

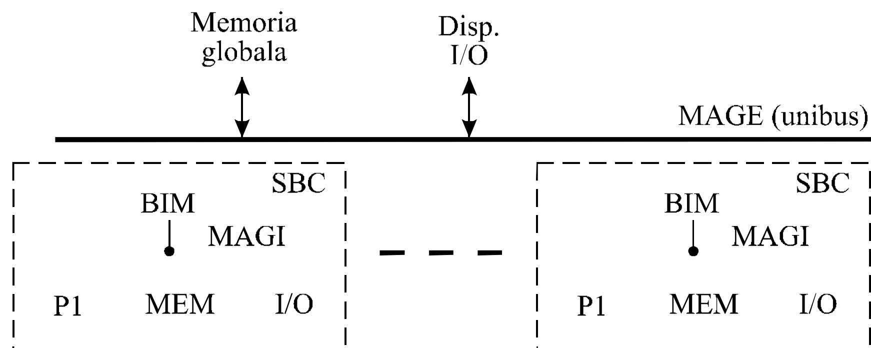


Figura 8.15. Interconectarea a doua SBC-uri

Din motive arhitecturale (blocaje, asteptari mari) si tehnologice, numarul de module de procesare (SBC) e limitat în practica la 30-32. O solutie pentru extinderea numarului de SBC-uri consta în interconectarea mai multor SMM-uri prin intermediul unor legaturi (seriale), rezultând structuri expandabile practic la orice dimensiune.

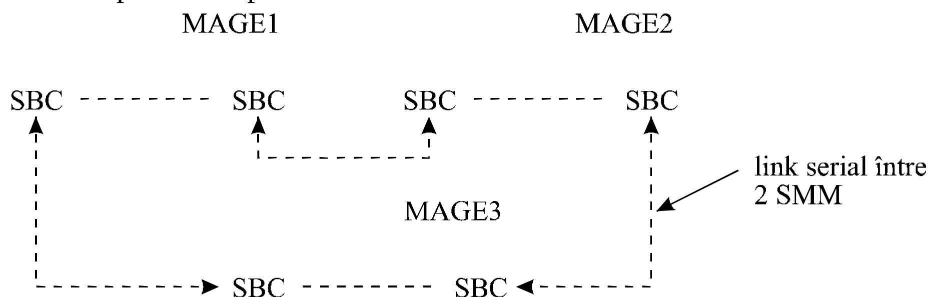


Figura 8.16. Interconectarea SMM-urilor

În acest caz, organizarea structurii se face pe baza conceptului de localizare al SBC-urilor, astfel:

- procesoarele cu schimburi mai frecvente de informatie sunt plasate pe aceeasi magistrala numita MAGE
- procesoarele cu frecventa de intercomunicare mai redusa vor comunica prin liniile seriale.

Exista 2 metode principiale de cuplare a busurilor MAGE pentru 2 SMM-uri:

a) Conectarea busurilor prin interfete de I/O seriale sau paralele

b) Conectarea strâns cuplata a busurilor prin tehnica Bus - Windows (BW)

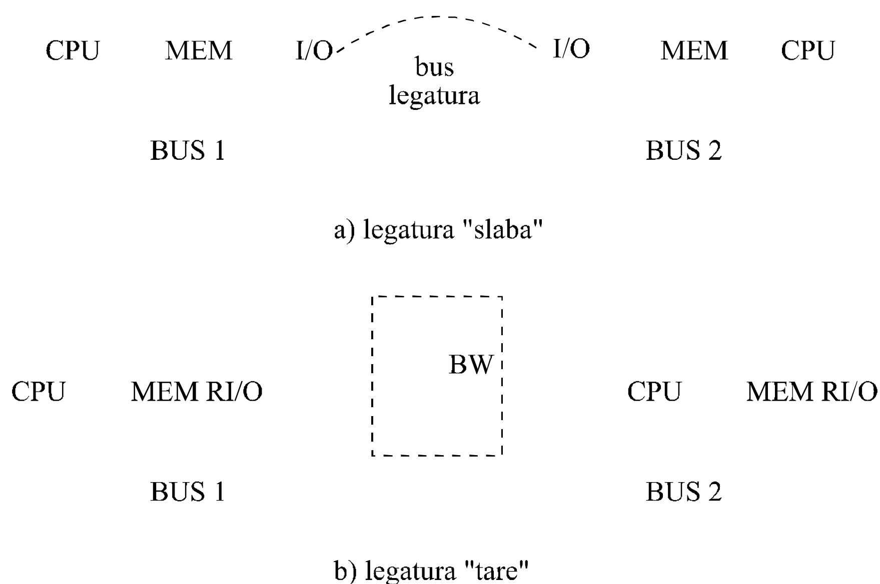


Figura 8.17. Legaturi slabe si tari

a. Sa presupunem ca un procesor de pe BUS1 doreste sa scrie într-o locatie de memorie de pe BUS2. Pentru asta, masterul pe BUS1 scrie în I/O de legatura un mesaj cu urmatorii parametrii: tip operatie, adresa, data, adresa procesor destinatie, sau DMA. I/O1 trimite aceste informatii pe busul de legatura la I/O2. Apoi I/O2 genereaza o întrerupere la procesorul destinatie sau DMA-ul respectiv care efectueaza operatia.

b. Secventa operatiilor succesive de acces este:
   1. interfata BW recunoaste ca un SLAVE2 este adresat pe bus1
   2. BW master, genereaza o cerere de BUS2
   3. Când accesul la BUS2 e acordat, prin BW se face transferul în mod transparent pentru masterul de pe BUS1

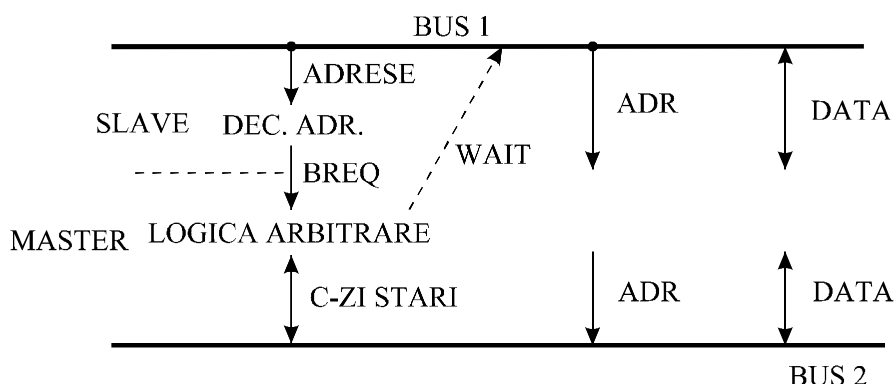


Figura 8.18. Modelul BW (Detaliu)

Esenta implementarii hardware a unui SMM consta în proiectarea unui sistem de comunicatie între MAGI a oricarui SBC si respectiv MAGE. O solutie consta în plasarea pe MAGE a unei memorii RAM comune, accesibile oricarui SBC, pe post de mailbox. Comunicatia între SBC-uri s-ar face în acest caz numai prin RAM-ul comun.

În acest caz, BIM-ul oricarui Pi va receptiona adresele de pe MAGI si va recunoaste adresa aferenta memoriei comune plasata pe MAGE ⇒ va lansa o cerere de bus (BUSRQ) pe MAGE ⇒ pune Pi în WAIT daca MAGE este ocupata sau relativ lenta.

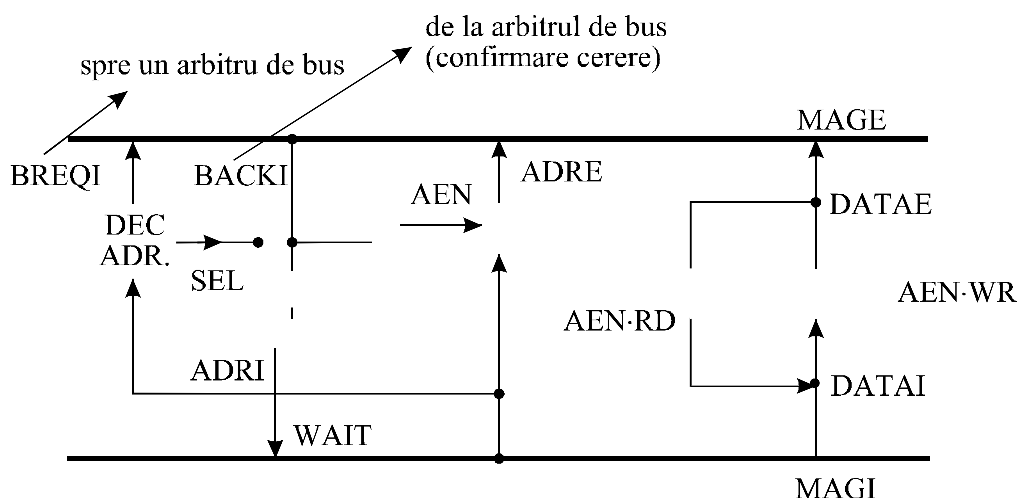


Figura 8.19. Accesarea magistralei externe

Pentru a rezolva cererile simultane de acces la resursa comuna din partea mai multor SBC-uri, exista un modul numit arbitru de bus (prioritati fixe sau rotitoare).

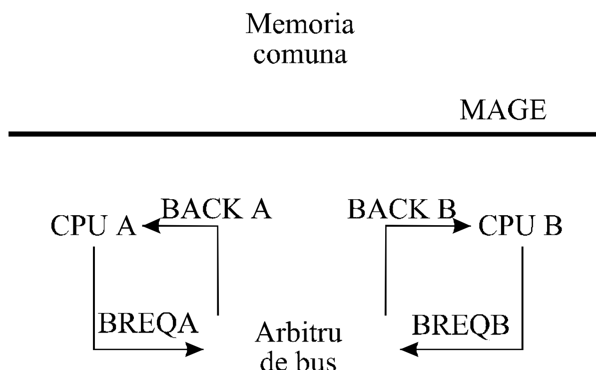


Figura 8.20. Arbitrarea magistralei

Solutia cu memorie comuna externa exclusiv concentrata pe MAGE prezinta dezavantajul dublarii timpului de ocupare al MAGE, pentru comunicatia CPUA $\rightarrow$ MEM $\rightarrow$ CPUB. O solutie care ar elimina acest dezavantaj consta în partajarea memoriei comune în 2 zone:

- memorie comuna externa concentrata, conectata fizic la MAGE, cu acces direct din partea oricarui procesor
- memorie comuna externa distribuita (D) în mod egal de toate procesoarele, accesata pe MAGI de propriul procesor si respectiv pe MAGE de oricare alt procesor (memorie biport).

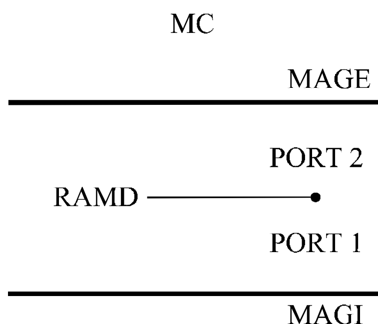


Figura 8.21. Comunicare prin intermediul memoriei biport

Este posibil ca memoria RAMD sa nu fie dublu port (accesibila MAGI, MAGE) ci doar uniport (MAGI) - memorie locala distribuita. În principiu, accesul la o memorie globala distribuita (dublu port) se face ca în figura urmatoare.

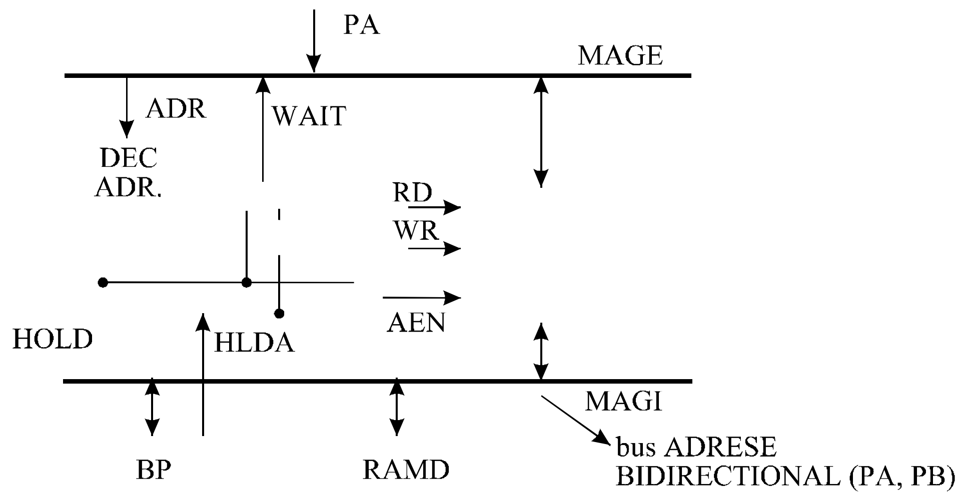


Figura 8.22. Accesul la memoria globala

PA pune pe MAGE o adresa ce corespunde accesarii memoriei RAMD. Decodificatorul de adresa sesizeaza si activeaza HOLD → PB ⇒ PA în WAIT pe MAGE. La finele ciclului de bus în curs, PB activeaza HLDA ⇒ WAIT inactiv ⇒ PA acceseaza RAMD datorita cuplarii MAGE ⇔ MAGI prin bufferele TS bidirectionale.

Clasificarea procesoarelor si memoriilor d.p.d.v. al adresabilitatii

Procesoare
a) P1=posibilitati de adresare MAGI, MAGE
b) P2=posibilitati de adresare numai pe MAGI

Memorii
a) M1=accesibila pe MAGI si MAGE (RAMD distribuita)
b) M2=accesibila exclusiv pe MAGI (locala)
c) m3=accesibila exclusiv pa MAGE (concentrata)

D.p.d.v. al SBC-urilor avem:
a) C1 ⇒ P1+M1 (eventual + M2)
b) C2 ⇒ P1+M2
c) C3 ⇒ P2+M1 (eventual + M2)

Prin combinarea acestor tipuri de SBC si memorii se obtin diverse arhitecturi caracteristice.

1) SBC C1

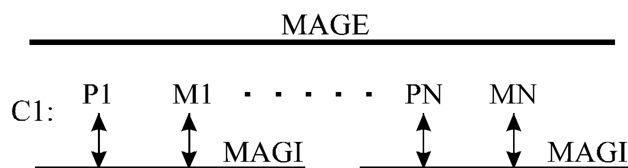


Figura 8.23. Modelul SBC C1

Prezinta flexibilitate si posibilitati multiple de comunicare între SBC-uri. M1 determina o scadere a traficului pe MAGE. Comunicarea între calculatoare se realizeaza prin memoriile M1 pe post de mailboxuri distribuite.

2) C1+C2

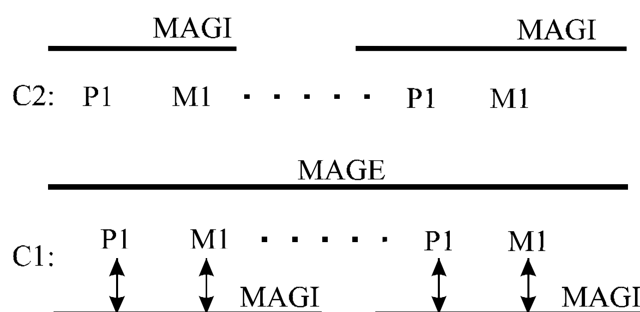


Figura 8.24. Modelul C1+C2

Comunicatia între C2 se poate face prin M1 din C1. Aceste comunicatii efectueaza negativ traficul pe MAGE .

3) C1+C3

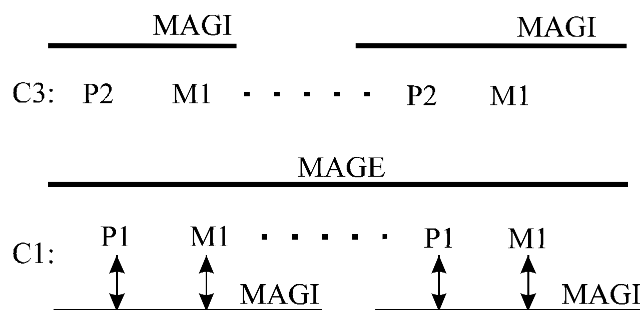


Figura 8.25. Modelul C1 + C3

C3 practic nu comunica între ele. Comunicatia C1-C3 se face prin M1 de pe C3. Aplicatia tipica pe un asemenea SMM consta în realizarea unor sisteme ierarhizate în care C3 realizeaza cuplarea la un proces condus.

4) C2+C3 (Memorie comuna centralizata)

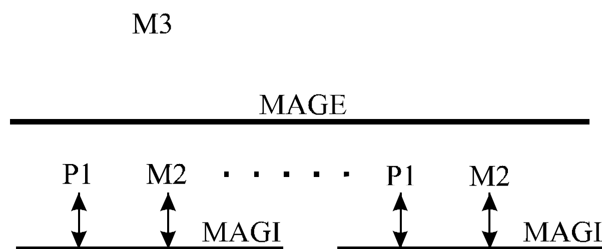


Figura 8.26. Modelul C2+C3

Comunicatia între oricare 2 SBC-uri se face prin intermediul memoriei comune M3 ⇒ încarcare mare a lui MAGE. Se aplica la SMM omogene.

8.9. TRANSPUTERE ÎN SMM

Transputerul (Transistor-Computer, TSP) este un μprocesor RISC pe 32 de biti, cu o arhitectura ce permite implementarea limbajului concurent OCCAM - considerat a fi limbajul "nativ" al TSP si SMM cu TSP-uri. TSP este construit ca element de baza al arhitecturilor paralele cu posibilitati evoluate de comunicare între procesele paralele. Asa de exemplu, TSP T-800 (INMOS-CO) detine 4 Ko SRAM intern, 4 canale externe de comunicatie intertransputere, procesor FPP etc.

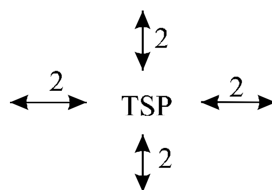


Figura 8.27. Link-urile unui TSP

Proiectarea TSP exploateaza disponibilitatea memoriei interne rapide, folosind doar 6 registri interni pentru executia proceselor secventiale:

- Workspace Pointer (WSP) - care pointeaza spre o zona de memorie unde sunt stocate variabilele locale ale procesorului activ curent.
- Operand register (OR) - care este utilizat în formarea operanzilor instructiunii.
- Registrul Instruction Pointer (PC) - pointeaza spre instructiunea urmatoare de executat.
- Registrii A,B,C - formeaza o stiva de evaluare (FIFO), reprezentând sursa respectiv destinatia pentru operatiile ALU. Instructiunile ALU se refera la ei în mod implicit. De exemplu o instructiune ADD, aduna cele 2 valori din vârful stivei (A). Se poate opera asupra acestor registri si prin instructiunea tip LOAD/STORE (set de instructiuni dupa modelul RISC).

Suportul pentru concurenta

TSP detine un planificator hardware μcodat care asigura un suport eficient pentru modelul OCCAM de concurenta si comunicare. În orice moment, un proces concurent poate fi:

- activ = daca este în executie (curent) sau în lista de asteptare a proceselor active.
- inactiv = daca este gata (pregatit) pentru comunicare (input/output) sau daca se afla într-o rutina de asteptare a unui anumit moment (eveniment).

Procesele active care asteapta sa fie executate sunt pastrate într-o lista înlantuita a spatiilor de lucru, implementata prin 2 registri: Front - pointeaza la zona de lucru (date) a primului proces din lista (P1) respectiv Back - care pointeaza la zona de date a ultimului proces din lista (Pn). Aceste zone de date memoreaza întreg contextul procesului respectiv, inclusiv PC-ul de revenire.

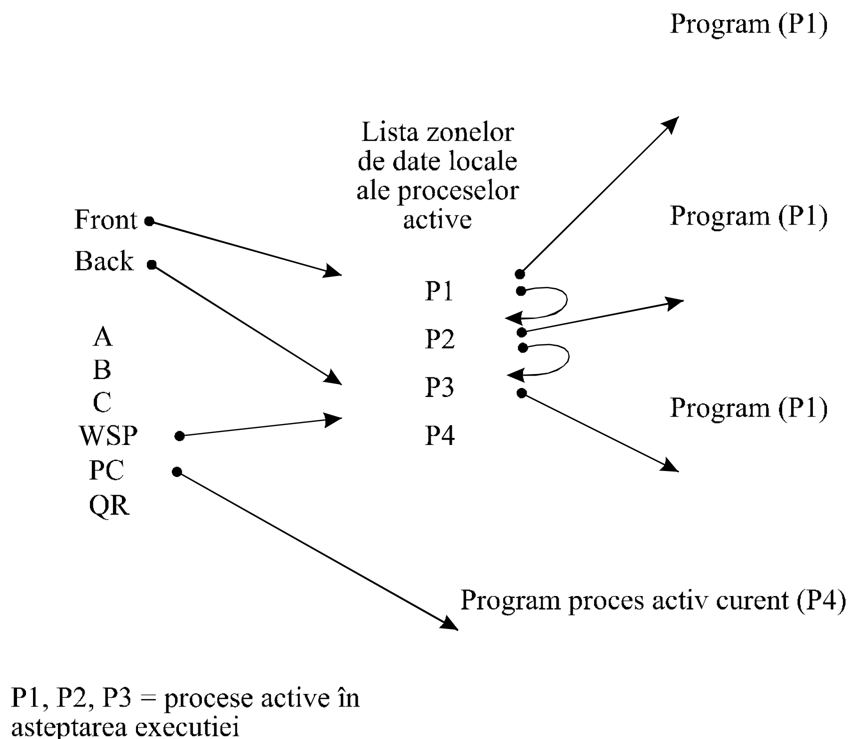


Figura 8.28. Comunicatia interprocese

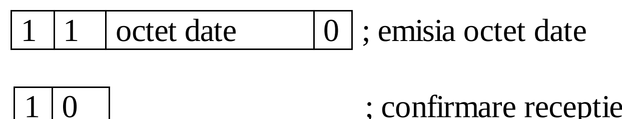
Un proces este executat pâna când devine inactiv. Când un proces ajunge inactiv, PC-ul sau este salvat în spatiul sau de lucru (Pk) si urmatorul proces din lista este activat curent.

Comunicatiile

Comunicatiile între procese sunt realizate cu ajutorul canalelor TSP. Un canal intern poate exista între 2 procese care se executa pe acelasi TSP si acest canal este implementat cu ajutorul unui singur cuvânt de memorie. Un canal extern poate exista între 2 procese care se executa pe TSP-uri diferite si este implementat printr-un canal serial bidirectional de comunicatie (din cele 4 ale TSP-ului). Conform modelului OCCAM, comunicatiile interprocese au loc atunci când atât procesul care face "output" cât si cel care face "input" sunt pregatite ("ready").

Un proces face un "input" sau "output" încarcând în stiva de evaluare: A=nr. de octeti de transferat, B=adresa canalului de comunicatie (un discriminator de adrese deduce intern sau extern), C=pointer la "mesajul" de transferat. Apoi, va executa o instructiune de tip IN/OUT mesaj. O legatura între 2 transputere este implementata prin conectarea unei interfete de

legatura a unui TSP cu cea a celuilalt TSP, prin 2 fire unidirectionale, prin OCCAM, unul în fiecare directie. Mesajele sunt transmise ca o secventa de biti cuprinzând un bit de start ('1'), un unu logic, opt biti de date si un bit de STOP ('0'). O confirmare este transmisa ca o secventa cuprinzând un bit de START si unul de STOP. Ea indica faptul ca procesul a fost capabil sa primeasca octetul de date si are loc în buffer pentru urmatorul.



Când un mesaj este pasat printr-un canal extern, TSP-ul da interfetei autonome de legatura sarcina de transfer a mesajului si scoate procesul din lista proceselor active. Când mesajul a fost transferat, interfata de legatura face procesorul sa replanifice procesul aflat cu executia celorlalte procese, în paralel cu transferul de mesaje (suprapunere executie/comunicare).

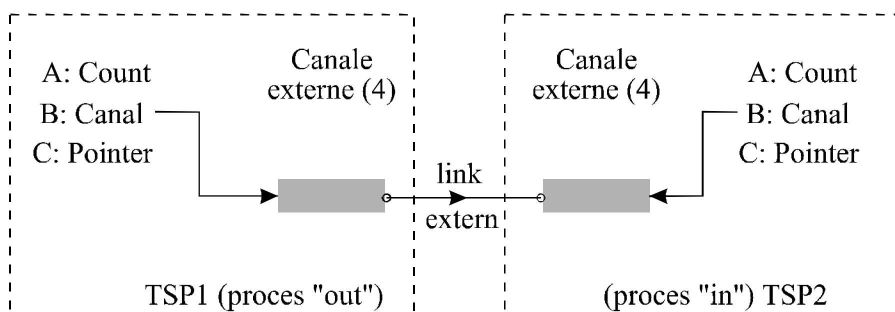


Figura 8.29. Transferul de mesaje prin canalul extern

Limbajul OCCAM – câteva aspecte

Permite scrierea unui program sub forma mai multor procese concurente (fire) executabile pe unul sau pe mai multe TSP-uri interconectate. Procesele concurente se vor distribui spre executie TSP-urilor din SMM prin intermediul links-urilor externe si vor putea fi procesate în paralel. Chiar daca aplicatia utilizeaza un singur TSP, programul poate fi construit ca un set de procese concurente care ar putea rula pe un anumit numar de TSP-uri. Acest stil de proiectare urmeaza cele mai bune traditii ale programarii structurate: procesele opereaza independent asupra variabilelor interne, cu exceptia cazurilor când interactioneaza explicit prin intermediul canalelor de comunicatie. Un set de procese

concurente poate rula pe un singur TSP sau, pentru marirea performantei, pot fi partitionate pe un numar oarecare de TSP-uri.

Toate programele OCCAM sunt construite pe baza a 3 procese primitive: asigurarea (unei variabile), intrarea si iesirea. Procesul de intrare atribuie unei variabile a programului valoarea citita de pe un canal de intrare.

Exemplu: chan 3? x înseamna ca variabila x primeste valoarea citita de pe canalul de intrare 3. Analog, procesul de iesire: chan 5 ! 4 ⇔ "întregul" 4 este emis pe canalul 5 de iesire.

Comunicatie prin intermediul unui canal nu poate starta pâna când ambele procese (in/out) nu sunt pregatite. Procesele primitive anterioare, pot fi grupate în asa-zise constructii în cadrul unui program OCCAM (SEQ, ALT, PAR).

Constructia SEQ

Specifica faptul ca procesele pe care le contine se vor executa secvential, în ordinea scrierii lor. Spre deosebire de limbajele secventiale, în OCCAM acest fapt trebuie specificat explicit.

Constructia PAR

Specifica faptul ca procesele pe care le contine se vor executa în paralel. Constructia PAR starteaza deci simultan procesele componente si se termina când toate procesele s-au terminat (sincronizare la bariera a proceselor înglobate).

Constructia ALT

În cadrul constructiei ALT care contine mai multe procese, se executa doar procesul al carui canal de intrare asociat devine activ primul. Dupa executia acestui proces constructia ALT se termina.

Exemplu:

```
ALT
        chan1 ? x
        proces 1        ; (Reglare "volum")
        chan2 ? x
        Proces 2        ; (Reglare "culoare")
        chan3 ? x
        Proces 3        ; (Modificare "canal")
        |
```

|
|

Comunicatia între procesoare

Reprezinta esenta programarii în OCCAM. În cazul cel mai simplu, sunt necesare 2 procesoare care se executa în paralel si un canal de legatura între ele. Comunicatia între procesele aferente unei constructii PAR, trebuie sa se faca numai prin intermediul canalelor - si nu atribuirilor - comunicatie. Urmând modelul OCCAM, comunicatiile au loc atunci când atât procesul care face "input" cât si cel care face "output", sunt gata. În consecinta, un proces care este gata trebuie sa astepte pâna când cel de al doilea devine la rândul sau gata.

Exemplu:

```
PAR
        INT x:
        SEQ
               input ? x
               com ! 2x
        INT y,z:
        SEQ
               z=z+5
               com ? y
               output ! y+1
```

Într-o constructie PAR, numai 2 procese pot folosi un anumit canal, unul pentru emisie, celalalt pentru receptie, în caz contrar rezultând un conflict pe canal. Trebuie avuta atentie astfel încât procesele din cadrul unei constructii PAR sa nu se astepte reciproc a.î. sa nu starteze nici unul ("deadlock"), ca în exemplul urmator.

Exemplu:

```
PAR
        SEQ
        com 1!2
        com 2?x
        SEQ
        com 2!3
        com 1?y
```

Obs. Solutia în acest caz consta în interschimbarea ordinii instructiunilor în cadrul uneia dintre cele 2 constructii SEQ.

Fiecare TSP detine un TIMER pentru implementarea prin soft a întârzierilor, pe diverse canale de timp (în general 1 ms). Un timer se comporta ca un canal intern, de pe care doar se receptioneaza (timpul curent) prin executia unei instructiuni "read timer". Un proces se poate pune în asteptare singur, executând instructiunea "timer input", caz în care va deveni disponibil pentru executie dupa atingerea unui anumit timp. Instructiunea "timer input" necesita specificarea unei valori a timpului. Daca aceasta valoare este în "trecut" (adica ClockReg AFTER TimpSpecificat) instructiunea nu are nici un efect (NOP). Daca însa timpul este în "viitor", atunci procesul este blocat. La atingerea timpului specificat procesul este planificat din nou. Exista de asemenea posibilitatea de a aloca canalele hardware externe la diverse canale logice (program) prin functia PLACE. Astfel de exemplu porturile de I/O pot fi adresate similar cu canalele:

<u>Exemplu:</u>

```
PORT OF BYTE
PLACE serial.status AT #3F8h:
INT STAT:
    |
    |
serial.status ? STAT
    |
    |
    |
```

Ca si în limbajele secventiale si în OCCAM exista procese repetitive (WHILE <cond>), procese conditionate (IF THEN ELSE), proceduri (PROC, proces având un nume asociat), functii (FUNCTION, proces apelat ce returneaza rezultate) etc.

8.10. ELEMENTE PRIVIND IMPLEMENTAREA SISTEMULUI DE OPERARE

Pe lânga resursele pe care le creeaza arhitectura SMM, sistemul de operare (S.O.) si aplicatia creeaza noi resurse si introduce ca urmare mecanisme de gestiune a acestora. Resursele sunt date de:

1) Procesele care apartin S.O. sau aplicatiei. Un proces reprezinta în fapt o pereche "program-context µprocesor", aflata în executie.
2) Procedurile, care implementeaza chiar functiile S.O. al SMM
3) Zone de date si variabile care apartin unui µprocesor (locale) sau sunt prezente în memoria comuna a SMM (partajate).

Mecanismele si componentele unui S.O. pe SMM implementeaza un numar de 3 functii care se refera la:

- sincronizare - reprezentata de necesitatea coordonarii operatiilor pe o resursa comuna
- excluderea mutuala - data de necesitatea utilizarii strict secventiale a unei resurse de catre procesele care intra în competitie pentru câstigarea ei
- comunicarea interprocese - cu conditia mentinerii coerentei variabilelor globale.

Mecanismele care implementeaza aceste functii sunt ierarhizate pe 2 nivele. Pe nivelul "inferior", acestea sunt responsabile cu coordonarea executiei proceselor, reprezentând mecanismele standard ale unui S.O. concurent. Pe nivelul "superior", functiile respective sunt responsabile cu coordonarea activitatii µprocesorului în cadrul aplicatiei. Evident, între cele 2 nivele exista interdependente strânse.

Exista un numar de 2 relatii fundamentale între procese:

a. Utilizarea unei resurse comune care nu a fost produsa de nici unul din procese
b. De "producator-consumator", în care un proces creeaza resurse care urmeaza sa fie utilizate si de alte procese. În cadrul acestei relatii apare necesitatea evitarii interblocajului ("deadlock"). Acesta poate sa apara în cazul în care procesele implicate se pun mutual în asteptare.

Pentru implementarea mecanismelor de gestiune, apar 3 concepte fundamentale:

1. Regiune critica (critical section) - desemnând o resursa (sectiune de cod) care poate fi controlata la un moment dat numai de catre un singur proces

2. Operatie atomica - secventa de cod neintreruptibila la nici un nivel
3. Semafor - constructie logica utilizata pentru controlul intrarii si iesirii dintr-o regiune critica, fiind implementat în general printr-o variabila numerica de stare.

În cadrul mecanismelor de comunicare si cooperare interprocese, este esentiala componenta S.O. numita monitor. Acesta este privit ca o entitate statica care este activata prin procesele care solicita accesul la resursele comune gestionate de monitor. Din acest p.d.v., monitorul reprezinta o regiune critica de cod si date cu rol de gestionare a accesului proceselor la un set de resurse comune conform unei anumite discipline de planificare. Pentru realizarea acestui scop, monitorul este alcatuit dintr-un set de proceduri si semafoare cu rol de gestionare a resursei pe care o monitorizeaza si de contorizare a referintelor la aceasta. Toate problemele legate de cooperarea proceselor sunt rezolvate într-un mod unitar în cadrul întregului SMM. Fiecare resursa din SMM are un monitor specific care functie de statutul resursei gestionate - locala sau partajata - poate fi rezident în memoria proprie a unui µprocesor sau în memoria comuna. Deci procesele nu pot opera direct pe resursa comuna - care în acest caz e reprezentata prin zone de date comune - ci numai prin intermediul procedurilor monitorului aferent. Mecanismul de gestiune implementat de monitor va fi dat de algoritmul folosit în utilizarea resursei si respectiv mecanismul de planificare a accesului proceselor la acea resursa.

9. PROBLEME PROPUSE SPRE REZOLVARE

1. Consideram un procesor scalar pipeline cu 5 nivele (IF, ID, ALU, MEM, WB) si o secventa de 2 instructiuni succesive si dependente RAW, în doua ipostaze:

A. i1: LOAD R_1, 9(R_5)
i2: ADD R_6, R_1, R_3

B. i1: ADD R_1, R_6, R_7
i2: LOAD R_5, 9(R_1)

a. Stabiliti cu ce întârziere (*Delay Slot*) starteaza a doua instructiune ?

b. Dar daca se aplica mecanismul de *forwarding* ?

În acest caz, pentru secventa **B**, cât ar fi fost întârzierea daca în cazul celei de a doua instructiuni, modul de adresare nu ar fi fost *indexat* ci doar *indirect registru* ? Comentati.

c. Verificati rezultatele obtinute pe procesorul DLX. Determinati cresterea de performanta obtinuta aplicând mecanismul de forwarding [($IR_{Cu\ forwarding}$ - $IR_{Fara\ forwarding}$) / $IR_{Fara\ forwarding}$].

Obs. Se considera ca operanzii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB.

2. Scrieti o secventa de program asamblare RISC care sa reprezinte translatarea corecta a programului scris în limbaj C ? Initial, registrii R_i, R_k, R_l, R_j, R_m contin respectiv variabilele i, k, l, j si m.

K = X[i-4]+12;
L = Y[j+5] XOR K;
M = K AND L;

Se considera programul executat pe un procesor scalar pipeline RISC cu 4 nivele (IF, ID, ALU/MEM, WB) si operanzii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB. Se cere:

a. Reprezentati graful dependentelor de date (numai dependentele de tip RAW).
b. În câte impulsuri de tact se executa secventa de program asamblare ?
c. Reorganizati aceasta secventa în vederea minimizarii timpului de executie (se considera ca procesorul detine o infinitate de registri generali).
d. Aplicând tehnica de *forwarding,* în câte impulsuri de tact se executa secventa reorganizata ?

3. Se considera o arhitectura superscalara caracterizata de urmatorii parametri (vezi **Figura 1:** ***Schema bloc a unei arhitecturi superscalare*** – L. Vintan, A. Florea – "*Sisteme cu microprocesoare – Aplicatii*", Editura Universitatii, Sibiu, 1999, pag. 189):

FR = 4 instr. / ciclu; – nr. instructiuni citite simultan din cache-ul de instructiuni sau memoria de instructiuni în caz de miss în cache.

IRmax = 2 (respectiv 4) instr. / ciclu; – nr. maxim de instructiuni independente lansate simultan în executie.

N_PEN = 10 impulsuri de tact; – nr. impulsuri de tact penalizare necesari accesului la memoria de instructiuni în caz de miss în cache.

Latenta = 2 impulsuri de tact; – nr. impulsuri de tact necesari executiei pentru orice tip de instructiune, lansata din buffer-ul de prefetch (unitati de executie nepipeline-izate).

IBS = 8 locatii (instructiuni); – dimensiunea buffer-ului de prefetch.

RmissIC = 40%; – rata de miss în cache-ul de instructiuni (din 5 citiri din IC, primele 2 sunt cu miss).

Consideram urmatoarea secventa succesiva de 20 instructiuni, caracterizata de hazarduri RAW aferente, executata pe arhitectura data.

i1 – i2 – ***i3 – RAW – i4*** – i5 ***– i6 – RAW – i7*** – i8 – i9 – i10 – ***i11 – RAW – i12*** – i13 – i14 – i15 – i16 – ***i17 – RAW – i18*** – i19 – i20. (în continuare “nu mai sunt instructiuni de executat”)

Obs. În cadrul unui ciclu de executie se realizeaza urmatoarele: din partea inferioara a buffer-ului de prefetch sunt lansate maxim IRmax instructiuni independente, iar simultan în partea superioara a buffer-ului sunt aduse, daca mai e spatiu disponibil, FR instructiuni din cache-ul sau memoria de instructiuni.

Determinati cresterea de performanta (studiu asupra ratei medii de procesare) prin varierea parametrului IRmax de la 2 la 4 instructiuni / ciclu. Prezentati în fiecare ciclu de executie continutul buffer-ului de prefetch.

4. Consideram un procesor RISC scalar pipeline caracterizat printre altele de urmatoarele instructiuni:

```
ADD     Ri, Rj, Rk – al doilea operand poate fi si valoare imediata
LD      Ri, adresa
ST      adresa, Ri
MOV     Ri, Rj
BEQ     Ri, Rj, label
BNE     Ri, Rj, label
J label
```

a. Acest procesor executa urmatoarea secventa:

```
ST      (R9), R6
LD      R10, (R9)
```

Rescrieti aceasta secventa folosindu-va de instructiunile cunoscute pentru a elimina **ambiguitatea referintelor la memorie** aparute în secventa originala data si a le executa mai rapid.

b. Se da secventa de instructiuni de mai jos:

```
ST      4(R5), R8
LD      R9, 8(R6)
```

Realizati o noua secventa cât mai rapida care sa înlocuiasca în mod corect pe cea de mai sus si care sa elimine posibila **ambiguitatea a referintelor la memorie** , favorizând executia instructiunii LD înaintea lui ST.

5. Dându-se urmatoarele secvente de instructiuni care implica dependente reale de date (RAW) sa se rescrie aceste secvente, cu un numar minim de instructiuni si folosind doar aceiasi registri (eventual R0 = 0 suplimentar), dar eliminând dependentele respective.

Obs. Unele instructiuni pot ramâne nemodificate.

a)
```
MOV   R6, R7
ADD   R3, R6, R5
```

b)
```
MOV   R6, #4
ADD   R7, R10, R6
LD    R9, (R7)
```

c)
```
MOV   R6, #0
ST    9(R1), R6
```

d)
```
MOV   R5, #4
BNE   R5, R3, Label
```

e)
```
ADD   R3, R4, R5
MOV   R6, R3
```

6. Se considera o arhitectura superscalara caracterizata de urmatorii parametri (vezi **Figura 1:** ***Schema bloc a unei arhitecturi superscalare*** – L. Vintan, A. Florea – "*Sisteme cu microprocesoare – Aplicatii*", Editura Universitatii, Sibiu, 1999, pag. 189):

FR = 4 (respectiv 8) instr. / ciclu; – nr. instructiuni citite simultan din cache-ul de instructiuni sau memoria de instructiuni în caz de miss în cache.

IRmax = 4 instr. / ciclu; – nr. maxim de instructiuni independente lansate simultan în executie.
N_PEN = 10 impulsuri de tact; – nr. impulsuri de tact penalizare necesari accesului la memoria de instructiuni în caz de miss în cache.
Latenta = 2 impulsuri de tact; – nr. impulsuri de tact necesari executiei pentru orice tip de instructiune, lansata din buffer-ul de prefetch (unitati de executie nepipeline-izate).
IBS = 8 locatii (instructiuni); – dimensiunea buffer-ului de prefetch.
RmissIC = 50% (initial); – rata de miss în cache-ul de instructiuni (din 2 citiri din IC, prima se va considera cu miss).

Pe arhitectura data se proceseaza urmatoarea secventa succesiva de 32 instructiuni, caracterizata de hazarduri RAW aferente.

i1 – i2 – ***i3 – RAW – i4*** – i5 ***– i6 – RAW – i7*** – i8 – i9 – i10 – ***i11 – RAW – i12*** – i13 – i14 – i15 – i16 – ***i17 – RAW – i18*** – i19 – i20 – i21 – ***i22 – RAW – i23*** – i24 – i25 – i26 – ***i27 – RAW – i28*** – i29 – i30 – ***i31 – RAW – i32***.

Obs. În cadrul unui ciclu de executie se realizeaza urmatoarele: din partea inferioara a buffer-ului de prefetch sunt lansate maxim IRmax instructiuni independente, iar simultan în partea superioara a buffer-ului sunt aduse, daca mai e spatiu disponibil, FR instructiuni din cache-ul sau memoria de instructiuni.

Determinati cresterea de performanta (studiu asupra ratei medii de procesare) prin varierea parametrului FR de la 4 la 8 instructiuni / ciclu. De mentionat ca aceasta variatie a FR de la 4 la 8 implica diminuarea ratei de miss în cache-ul de instructiuni cu 50%. Prezentati în fiecare ciclu de executie continutul bufferului de prefetch. În ce consta limitarea performantei în acest caz ?

7. Se considera urmatoarea secventa de instructiuni executata pe un procesor pipeline cu 4 nivele (IF, ID, ALU/MEM, WB), fiecare faza necesitând un tact, cu urmatoarea semnificatie:

IF = citirea instructiunii din cache-ul de instructiuni sau din memorie
ID = decodificarea instructiunii si citirea operanzilor din setul de registri generali
ALU / **MEM** = executie instructiuni aritmetice sau accesare memorie

WB = înscriere rezultat în registrul destinatie

Operanzii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB.

i1: ADD R_i, R_0, #i
i2: ADD R_j, R_i, #4
i3: LOAD R_1, (R_i)
i4: LOAD R_2, (R_j)
i5: ADD R_3, R_1, R_2
i6: SUB R_4, R_1, R_2
i7: ABS R_4, R_4
i8: ADD R_1, R_{13}, R_{14}
i9: DIV R_1, R_1, #2
i10: STORE (R_i), R_1

Se cere:

a. Reprezentati graful dependentelor de date (RAW, WAR, WAW) si precizati în câte impulsuri de tact se executa secventa? Initial, structura "*pipe*" de procesare este goala, registri initializati cu 0.

b. Reorganizati aceasta secventa în vederea minimizarii timpului de executie (se considera ca procesorul detine o infinitate de registri generali disponibili). În câte impulsuri de tact s-ar procesa în acest caz secventa ?

c. Ce simuleaza secventa initiala daca în instructiunea i8 în locul registrilor R_{13} am avea R_3 si în locul lui R_{14} am avea R_4. Precizati formula matematica obtinuta.

8. Se considera secventa de program RISC:

i1: ADD R_1, R_2, #15
i2: ADD R_3, R_4, #17
i3: ADD R_5, R_3, R_1
i4: ADD R_6, R_5, #12
i5: ADD R_3, R_7, #3
i6: ADD R_8, R_3, #2
i7: ADD R_9, R_8, #14

Se cere:

a. Sa se construiasca graful dependentelor de date (RAW, WAR, WAW) aferent acestei secvente si precizati în câte impulsuri de tact

se executa secventa, stiind ca latenta de executie a instructiunii ADD este de 1 ciclu ?

b. Sa se determine modul optim de executie al acestei secvente reorganizate, pe un procesor RISC superscalar cu 6 seturi de registri generali si 3 unitati ALU.

9. a) Are sens implemetarea unui bit de validare (V) pentru fiecare bloc din cache în cadrul unui sistem de calcul uniprocesor (fara DMA) ? Justificati.

b) O exceptie *Page Fault* implica întotdeauna una de tip TLB miss ? Dar reciproc ?

10. a) Ce limitare principiala exista asupra ratei de fetch a instructiunilor ? Cum ar putea fi depasita aceasta limitare ?

b) Descrieti succint tehnicile de eliminare a dependentelor RAW între instructiuni.

c) Daca într-o arhitectura tip Tomasulo ar lipsi câmpurile de "*tag*" (Q_j,Q_i) din cadrul statiilor de rezervare, considerati ca ar mai functiona schema ? Justificati punctual.

11. Se considera un microprocesor RISC cu o structura «pipe» de procesare a instructiunilor, având vectorul de coliziune atasat 101011. Sa se determine rata teoretica optima de procesare a instructiunilor pentru acest procesor [instr/ciclu] si sa se expliciteze algoritmul dupa care trebuie introduse instructiunile în structura.

12. Fie vectorul de coliziune 1001011 atasat unui microprocesor RISC cu o structura «pipe» de procesare a instructiunilor. Sa se determine rata teoretica optima de procesare a instructiunilor pentru acest procesor [instr/ciclu] si explicitând algoritmul dupa care trebuie introduse instructiunile în structura. Determinati cresterea de performanta comparativa cu situatia în care s-ar procesa pe o structura CISC conventionala [$(IR_{RISC} - IR_{CISC}) / IR_{CISC}$].

13.a) În cadrul unui procesor vectorial se considera urmatoarea secventa de program:

```
x = 0
for i = 1 to 100 do
```

```
        x = x + A[i]*B[i];
    endfor
```

În câti cicli de tact se executa secventa ? Este bucla vectorizabila ? În caz negativ, scrieti o noua secventa de program care sa aiba acelasi efect dar care sa fie executata mult mai rapid ? Determinati noul timp de executie al secventei ? Concret din ce motive se câstiga timp de procesare ?

b) Sa se proiecteze un cache de instructiuni cuplat la un procesor superscalar (VLIW). Lungimea blocului din cache se considera egala cu rata de fetch a procesorului, în acest caz 8 instructiuni / bloc. Cache-ul va fi de tipul **4 way set associative** (cu 4 blocuri / set). Explicati semnificatia fiecarui câmp utilizat (necesar) în proiectare.

c) Descrieti avantajelor introduse de Tomasulo în cadrul arhitecturii care îi poarta numele.

14.a) Prezentati pasii succesivi aferenti drumului de la o arhitectura parametrizabila data, la instanta sa optimala, numita procesor.

b) De ce nu este suficienta doar optimizarea unitatilor secventiale de program ("basic-blocks"), fiind necesara optimizarea globala a întregului program?

c) Care sunt în opinia dvs. motivele pentru care microprocesoarele superscalare au un succes comercial net superior celor cu paralelism exploatat prin optimizari de program ("Scheduling" static) – de exemplu microprocesoare tip VLIW, EPIC, etc. ?

15. Relativ la gestiunea memoriei în sistemele moderne, tratati într-o maniera concisa urmatoarea problematica:

a. În ce rezida necesitatea unui sistem de memorie virtuala (MV) ?
b. Explicati principiile protectiei în sistemele cu MV ?
c. Explicati succint rolul TLB (Translation Lookaside Buffer) în translatarea adreselor.
d. În ce consta dificultatile implementarii unui cache virtual ? Care ar fi avantajele acestuia ?

16.a) Caracteristici ale structurilor de program care limiteaza accelerarea unui sistem paralel de calcul în raport cu unul secvential;

b) Tipuri arhitecturale de memorii cache. Avantajele/dezavantajele fiecarei organizari. Explicitati succint notiunea de coerenta a cache-urilor în sistemele multiprocesor;
c) Limitari arhitecturale ale paradigmei superscalare si independente de masina, referitoare la:
   - maximizarea *ratei de aducere a instructiunilor din memorie*
   - maximizarea *ratei de executie a instructiunilor.*

17.a) Care este semnificatia termenilor:
- Arhitectura RISC (Reduced Instruction Set Computer)
- Arhitectura CISC (Complex Instruction Set Computer)

Care arhitectura este mai rapida si de ce ?

b) Explicati semnificatia nivelelor pipeline aferente unui microprocesor (IF, ID, ALU, MEM, WB). Explicati avantajele introduse de conceptul de procesare "pipeline".
c) Ce este o arhitectura VLIW (Very Large Instruction Word) ? Descrieti asemanarile si deosebirile dintre o arhitectura VLIW si una superscalara. Subliniati avantajele arhitecturii VLIW fata de o arhitectura conventionala de procesor.
d) Un microprocesor cu 4 nivele pipeline (IF, ID, ALU/MEM, WB) executa urmatoarea secventa de cod:

```
        ADD         ; Adunare
        BR   SUB1   ; Salt neconditionat - apel subrutina
        SUB         ; Scadere
        MUL         ; Înmultire
        .
        .
SUB1:                ; Intrarea în subrutina - adresa primei
                     ; instructiuni din subrutina
```

Ce se întâmpla când procesorul întâlneste instructiunea de salt (situatia instructiunilor în "pipe")? Considerând ca procesorul primeste o întrerupere dupa ce executa instructiunea de salt, explicati clar secventa de evenimente petrecute dupa receptia întreruperii.

18. Consideram un procesor pipeline cu 5 nivele (IF, ID, ALU, MEM, WB) în care conditia de salt este verificata pe nivelul de decodificare, operanzii instructiunilor sunt necesari la finele fazei ID, iar rezultatele sunt disponibile în setul de registri generali la finele fazei WB. Trasati diagrama ciclului de tact a procesorului incluzând dependentele reale de

date (RAW), reprezentati graful dependentelor de date, la executia urmatoarei secvente de instructiuni. Stabiliti cu ce întârziere (*Delay Slot*) starteaza a doua instructiune în cazul existentei unui hazard RAW între doua instructiuni; care este timpul total de procesare al secventei. Explicit subliniati toate tipurile de hazard (de date, de ramificatie) si daca si unde poate fi aplicat mecanismul de forwarding. Presupunând ca instructiunea de salt este corect predictionata ca *non-taken* în câti cicli de tact se va procesa secventa ?

```
ADD  R1, R2, R3
LD   R2, 0(R1)
BNE  R2, R1, dest
SUB  R5, R1, R2
LD   R4, 0(R5)
SW   R4, 0(R6)
ADD  R9, R5, R4
```

19.a) În general un microprocesor consuma multe impulsuri de tact pentru a citi o locatie de memorie DRAM, datorita latentei relativ mari a acesteia. Implementarea caror concepte arhitecturale micsoreaza timpul mediu de acces al microprocesorului la memoria DRAM ?

b) Care este principalul concept arhitectural prin care se micsoreaza timpul mediu de acces al procesorului la discul magnetic ?

20.a) Care sunt principalele cauze ce limiteaza performanta unei structuri "*pipeline*" de procesare a instructiunilor masina ?

b) De ce timpul de executie al unui program pe un multiprocesor cu N procesoare nu este în general de N ori mai mic decât timpul de executie al aceluiasi program pe un sistem uniprocesor ?

c) La ce se refera necesitatea coerentei memoriilor cache dintr-un sistem multiprocesor ? Ce strategii principiale de mentinere a coerentei memoriilor cache cunoasteti ?

21. Sa se proiecteze un cache de instructiuni cuplat la un procesor superscalar (VLIW). Lungimea blocului din cache se considera egala cu rata de fetch a procesorului, în acest caz 4 instructiuni / bloc. Cache-ul va fi de tipul:

a) semiasociativ, cu 2 blocuri / set (2 – way set associative)
b) complet asociativ (full - associative)
c) cu mapare directa (direct mapped)

Ce se întâmpla daca în locul adresarii cu adrese fizice se considera adresare cu adresa virtuala?

22. a) De ce este dificila procesarea «Out of Order» a instructiunilor Load respectiv Store într-un program si de ce ar putea fi ea benefica?

b) Care dintre cele doua secvente de program s-ar putea procesa mai rapid pe un procesor superscalar cu executie «Out of Order» a instructiunilor? Justificati.

B1.

```
for i=1 to 100
  a[2i]=x[i];
  y[i]=a[i+1];
```

B2.

```
a[2]=x[1];
y[1]=a[2]+5;
for i=2 to 100
```

```
a[2i]=x[i];

y[i]=a[i+1]+5;
```

23. Se considera un procesor scalar pipeline, în 3 faze diferite de procesare (IF,EX,WR), fiecare faza necesitând un tact, cu urmatoarea semnificatie:

IF = aducere si decodificare a instructiunii
EX=selectie operanzi din setul de registri si executie
WR=înscriere rezultat în registrul destinatie

Se considera secventa de program:

```
1: R1<-- (R11)+(R12)
2: R1<-- (R1)+(R13)
3: R2 <-- (R3)+4
4: R2 <-- (R1)+(R2)
5: R1<-- (R14)+(R15)
6: R1<-- (R1)+(R16)
```

a) În câte impulsuri se executa secventa? (initial, structura «pipe» de procesare este «goala») Reorganizati aceasta secventa de program în vederea minimizarii timpului de executie (procesorul detine o infinitate de registri generali disponibili). În câte impulsuri de tact s-ar procesa în acest caz secventa ?

b) În câte tacte (minimum) s-ar procesa secventa daca procesorul ar putea executa simultan un numar nelimitat de instructiuni independente? Se considera ca procesorul poate aduce în acest caz, simultan, 6 instructiuni din memorie. Justificati.

24. Se considera o structura «pipe» de procesare a instructiunilor având un nivel de citire a operanzilor din setul de registri (RD), situat anterior unui nivel de scriere a rezultatului în setul de registri (WR). Careia dintre cele doua operatii (RD, WR) i se da prioritate în caz de conflict si în ce scop ?

25. Se considera ca 20% dintre instructiunile unui program determina ramificarea acestuia (salt efectiv). Care ar fi în acest caz rata de fetch (FR) posibila pentru un procesor superscalar (VLIW – Very Long Instruction Word) având resurse harware nelimitate si o predictie perfecta a branch-urilor (cunoastere anticipata a adresei de salt) ? Este posibila o depasire a acestei limitari fundamentale ? Daca da, care ar fi noua limitare impusa parametrului FR prin solutia Dvs. ?

26. Un procesor superscalar poate lansa în executie simultan maxim N instructiuni ALU independente. Logica de detectie a posibilelor hazarduri RAW (Read After Write) între instructiunile ALU are costul «C» ($). Cât va costa logica de detectie daca s-ar dori ca sa se poata lansa simultan în executie maxim (N+1) instructiuni ALU independente ? (Se vor considera costurile ca fiind direct proportionale cu «complexitatea» logicii de detectie a hazardurilor RAW).

27. Relativ la o memorie cache cu mecanism de adresare tip «mapare directa», precizati valoarea de adevar a afirmatiilor de mai jos, cu justificarile de rigoare.

a) Rata de hit creste daca capacitatea memoriei creste;
b) data de la o anumita locatie din memoria principala poate fi actualizata la orice adresa din cache;
c) Scrieri în cache au loc numai în ciclurile de scriere cu miss în cache;
d) Are o rata de hit net mai mare decât cea a unei memorii complet asociative si de aceeasi capacitate.

28. Se considera secventa de program RISC:

```
1:   ADD   R1, R11, R12
2:   ADD   R1, R1, R13
3:   ADD   R2, R3, R9
4:   ADD   R2, R1, R2
5:   ADD   R1, R14, R15
```

a) Reprezentati graful dependentelor de date (numai dependentele de tip RAW)
b) Stiind ca între 2 instructiuni dependente RAW si succesive e nevoie de o întârziere de 2 cicli, în câti cicli s-ar executa secventa ?
c) Reorganizati secventa în vederea unui timp minim de executie (nu se considera alte dependente decât cele de tip RAW).

29.a) Considerând un procesor RISC pe 5 nivele pipe (IF, ID, ALU, MEM, WB), fiecare durând un ciclu de tact, precizati câti cicli de întârziere («branch delay slot») impune o instructiune de salt care determina adresa de salt la finele nivelului ALU ?
b) De ce se prefera implementarea unor busuri si memorii cache separate pe instructiuni, respectiv date în cazul majoritatii procesoarelor RISC (pipeline) ?
c) De ce sunt considerate instructiunile CALL / RET mari consumatoare de timp în cazul procesoarelor CISC (ex. I-8086) ? Cum se evita acest consum de timp în cazul microprocesoarelor RISC ?

30. Considerând un microprocesor virtual pe 8 biti, având 16 biti de adrese, un registru A pe 8 biti, un registru PC si un registru index X, ambele pe 16 biti si ca opcode-ul oricarei instructiuni e codificat pe 1 octet, sa se determine numarul impulsurilor de tact necesare aducerii si executiei instructiunii «memoreaza A la adresa data de (X+deplasament)». Se considera ca instructiunea e codificata pe 3 octeti si ca orice procesare (operatie interna) consuma 2 tacte. Un ciclu de fetch opcode dureaza 6 tacte si orice alt ciclu extern dureaza 4 tacte.

31. Relativ la o arhitectura de memorie cache cu mapare directa se considera afirmatiile:

a) Nu permite accesul simultan la câmpul de date si respectiv «tag» al unui cuvânt accesat.

b) La un acces de scriere cu hit, se scrie în cache atât data de înscris cât si «tag-ul» aferent.
c) Rata de hit creste usor daca 2 sau mai multe blocuri din memoria principala - accesate alternativ de catre microprocesor - sunt mapate în acelasi bloc din cache.

Stabiliti valoarea de adevar a acestor afirmatii si justificati pe scurt raspunsul.

32. Ce corectie (doar una!) trebuie facuta în secventa de program asamblare pentru ca translatarea de mai jos sa fie corecta si de ce ? Initial, registrii R_i, R_k, R_l, R_j contin respectiv variabilele i, k, l, j. Primul registru dupa mnemonica este destinatie. (R_j+offset) semnifica operand în memorie la adresa data de (R_j+offset).

k = a[i+2]+5;	i1:	ADD	R_k, #2, R_i
l = c[j+9] - k;	i2:	LOAD	R_k, (R_k+0)
	i3:	ADD	R_k, R_k, #5
	i4:	ADD	R_l, #9, R_j
	i5:	LOAD	R_l, (R_j+0)
	i6:	SUB	R_l, R_l, R_k

33. Se considera un microsistem realizat în jurul unui microprocesor care ar accepta o frecventa maxima a tactului de 20 MHZ. Regenerarea memoriei DRAM se face <u>în mod transparent</u> pentru microprocesor. Procesul de regenerare dureaza 250 ns. Orice ciclu extern al procesorului dureaza 3 perioade de tact. Poate functiona în aceste conditii microprocesorul la frecventa maxima admisa? Justificati.

34. Explicati concret rolul fiecareia dintre fazele de procesare (ALU, MEM, WB) în cazul instructiunilor:

a) STORE <u>R5</u>, (R9)06h;
sursa

b) LOAD <u>R7</u>, (R8)F3h;
dest

c) AND <u>R5</u>, R7, R8.

dest

35. Se considera un procesor scalar pipeline, în 3 faze diferite de procesare (IF, EX, WR), fiecare faza necesitând un tact, astfel:

IF = fetch instructiune si decodificare;
EX = selectie operanzi din setul de registri si executie;
WB = înscriere rezultat în registrul destinatie.

a) În câte impulsuri de tact se executa secventa de program de mai jos ?
b) Reorganizati aceasta secventa în vederea minimizarii timpului de executie.

```
1:    ADD   R3, R2, #2
2:    ADD   R1, R9, R10
3:    ADD   R1, R1, R3
4:    ADD   R2, R3, #4
5:    ADD   R2, R1, R2
6:    STORE R3, (R1)2
```

36. Un procesor pe 32 biti la 50 MHZ, lucreaza cu 3 dispozitive periferice prin interogare. Operatia de interogare a starii unui dispozitiv periferic necesita 100 de tacte. Se mai stie ca:

a) interfata cu mouse-ul trebuie interogata de 30 de ori / s pentru a fi siguri ca nu se pierde nici o «miscare» a utilizatorului.
b) floppy - discul transfera date spre procesor în unitati de 16 biti si are o rata de transfer de 50 ko / s.
c) hard - discul transfera date spre procesor în unitati de 32 biti si are o rata de transfer de 2 Mo / s.

Determinati în [%], fractiunea din timpul total al procesorului, necesara interogarii starii fiecarui periferic. Comentati.

37. Se considera instructiunea (I-8086):

<u>3000h</u>: MOV <u>[BX]0F3h</u>, <u>AX</u>
EA dest sursa

a) La ce adresa fizica se aduce opcode-ul instructiunii ?

b) La ce adrese fizice se scriu registrii AL, respectivAH ?
Înainte de executia instructiunii avem: CS = 1D00h
BX = 1B00h
SS = 2000h
DS = DF00h.

38. Se considera instructiunea (I-8086):

2000h: PUSH AX
EA

a) De la ce adresa se aduce instructiunea ?
b) La ce adrese fizice se scriu registrii AL, respectiv AH ?

Se stie ca înainte de executia instructiunii PUSH avem:CS = AE00h
SS = 1FF0h
SP = 001Eh
DS = 1F20h.

39. Un automat de regenerare al memoriilor DRAM declanseaza efectiv procesul de regenerare daca sunt simultan îndeplinite conditiile:

a) activare semnal cerere refresh (CREF).
b) microprocesorul nu lucreaza momentan cu memoria. Având în vedere dezideratul regenerarii «transparente» (sa nu fie simtita de catre microprocesor), ar functiona corect automatul ? Comentati si sugerati o eventuala corectie.

40. Consideram 3 memorii cache care contin 4 blocuri a câte un cuvânt / bloc. Una este complet asociativa, alta semiasociativa cu 2 seturi a câte 2 cuvinte si ultima cu mapare directa. Stiind ca se foloseste un algoritm de evacuare de tip LRU, determinati numarul de accese cu HIT pentru fiecare dintre cele 3 memorii, considerând ca procesorul citeste succesiv de la adresele 0, 8, 0, 6, 8, 10, 8 (primul acces la o anumita adresa va fi cu MISS).

41. Se considera secventa de program RISC:

```
1:   ADD  R3, R2, #2
2:   ADD  R1, R9, R10
3:   ADD  R1, R1, R3
```

4: ADD R2, R3, #4
5: ADD R2, R1, R2

Între doua instructiuni dependente RAW si succesive în procesare, e nevoie de o întârziere de 1 ciclu de tact.

a) În câti cicli de tact se executa secventa initiala ?

b) În câti cicli de tact se executa secventa reorganizata aceasta secventa în vederea unui timp minim de procesare ?

42. Se considera o unitate de disc având rata de transfer de 25×10^4 biti/s, cuplata la un microsistem. Considerând ca transferul între dispozitivul periferic si CPU se face prin întrerupere la fiecare octet, în mod sincron, ca timpul scurs între aparitia întreruperii si intrarea în rutina de tratare este de 2µs si ca rutina de tratare dureaza 10µs, sa se calculeze timpul pe care CPU îl are disponibil între 2 transferuri succesive de octeti.

43. a. Daca rata de hit în cache ar fi de 100%, o instructiune s-ar procesa în 8.5 cicli de tact. Sa se exprime în [%] scaderea medie de performanta daca rata de hit devine 89%, orice acces la memoria principala se desfasoara pe 6 tacte si ca orice instructiune face 3 referinte la memorie.

b. De ce e avantajoasa implementarea unei pagini de capacitate «mare» într-un sistem de memorie virtuala ? De ce e dezavantajoasa aceasta implementare ? Pe ce baza ar trebui facuta alegerea capacitatii paginii ?

44. Se considera un procesor scalar pipeline, în 3 faze diferite de procesare (IF, EX, WR), fiecare faza necesitând un tact, astfel:

IF = fetch instructiune si decodificare;
EX = selectie operanzi din setul de registri si executie;
WB = înscriere rezultat în registrul destinatie.

a) În câte impulsuri de tact se executa secventa de program de mai jos ?

b) Reorganizati aceasta secventa în vederea minimizarii timpului de executie (se considera ca procesorul detine o infinitate de registri generali).

1: R1 ← (R11) + (R12)
2: R1 ← (R1) + (R13)
3: R2 ← (R3) + 4
4: R2 ← (R1) + (R2)
5: R1 ← (R14) + (R15)
6: R1 ← (R1) + (R16)

45. Se considera un microprocesor RISC cu o structura «pipe» de procesare a instructiunii, având vectorul de coliziune atasat 01011. Sa se determine rata teoretica optima de procesare a instructiunii pentru acest procesor [instr/ciclu].

46. De ce implementarea algoritmului lui R. TOMASULO într-o arhitectura superscalara ar putea reduce din «presiunea» la citire asupra seturilor de registri generali ? Gasiti vreo similitudine în acest sens, între un CPU superscalar având implementat acest algoritm si un CPU de tip TTA (Transport Triggered Architecture) ?

47. De ce considerati o instructiune de tip RETURN este mai dificil de predictionat printr-un predictor hardware ? Puteti sugera vreo solutie în vederea eliminarii acestei dificultati ? În ce consta noutatea «principiala» a predictoarelor corelate pe doua nivele ?

48. Cum credeti ca s-ar putea masura printr-un simulator de tip «trace driven», câstigul de performanta introdus de tehnicile de paralelizare a buclelor de program (ex. «Loop Unrolling», «Software Pipelining», etc.)

49. Cum explicati posibilitatea interblocarii proceselor în cadrul limbajului OCCAM ? Ce întelegeti prin «sectiune critica de program» în cadrul unui sistem multimicro ? Care este «mesajul» transmis de «legea lui AMDAHL» pentru sistemele paralele de calcul ?

50. Se considera structura hardware a unui microprocesor RISC, precum în figura de mai jos.

Raspundeti la urmatoarele întrebari.

a) Ce tip de instructiuni activeaza sumatorul «sum 2» si în ce scop ?

b) Într-un tact, la setul de registri pot fi necesare 2 operatii simultane: citire (nivelul RD din pipe), respectiv scriere (nivelul WB din pipe). Carei operatii i se da prioritate si în ce scop ?

c) Ce rol are unitatea ALU în cazul unei instructiuni de tip LOAD ?

d) Ce informatie se memoreaza în latch-ul EX/MEM în cazul instructiunii: ST (R7)05, R2 si de unde provine fiecare informatie ?

51. Se considera secventa de program RISC:

```
1:    SUB   R7, R2, R12
2:    ADD   R1, R9, R10
3:    ADD   R1, R1, R7
4:    SUB   R2, R7, R12
5:    ADD   R2, R1, R2
6:    ADD   R1, R6, R8
7:    ADD   R1, R1, R7
8:    SUB   R1, R1, R12
9:    LD    R1, (R1)2
10:   LD    R4, (R4)6
11:   ADD   R1, R4, R1
12:   ADD   R1, R1, R2
13:   ST    R1, (R4)16
14:   ST    R7, (R1)16
```

a) Sa se construiasca graful dependentelor / precedentelor de date aferent acestei secvente. Cu exceptia LOAD-urilor care au latenta de 2 cicli, restul instructiunilor au latenta de 1 ciclu.
b) În baza algoritmului LIST SCHEDULING, sa se determine modul optim de executie al acestei secvente (nr. cicli), pentru un procesor superscalar având 2 unitati ADD, 1 unitate SUB si 1 unitate LOAD/STORE. Unitatea pentru LOAD este nepipeline-izata.

În limbaj de asamblare MIPS prezentati solutiile urmatoarelor probleme:

52. Scrieti un program folosind recursivitatea, care citeste caractere de la tastatura si le afiseaza în ordine inversa.

Obs. Nu se lucreaza cu siruri, nu se cunoaste numarul de caractere citite, sfârsitul sirului va fi dat de citirea caracterului '0'.

Modificati programul astfel încât '0' – care marcheaza sfarsitul sirului, sa nu fie tiparit.

53. Scrieti un program folosind recursivitatea, care citeste de la tastatura doua numere întregi pozitive si afiseaza cel mai mare divizor comun si cel mai mic multiplu comun al celor doua numere.

54. Scrieti un program recursiv care rezolva problema Turnurilor din Hanoi pentru n discuri (n – parametru citit de la tastatura). Enuntul problemei este urmatorul:

Se dau trei tije simbolizate prin A, B si C. Pe tija A se gasesc n discuri de diametre diferite, asezate în ordine descrescatoare a diametrelor privite de jos în sus. Se cere sa se mute discurile de pe tija A pe tija B, folosind tija C ca tija de manevra, respectându-se urmatoarele reguli:

- ❖ La fiecare pas se muta un singur disc.
- ❖ Nu este permis sa se aseze un disc cu diametrul mai mare peste un disc cu diametrul mai mic.

55. Realizati un program care citeste de la tastatura doua numere naturale n si k (n>k) si calculeaza si afiseaza pe consola valorile urmatoare:

C_n^k si A_n^k

56. Sa se citeasca un sir de numere întregi de la tastatura, a carui dimensiune este citita tot de la tastatura. Sortati sirul prin metoda “bubblesort”, memorati succesiv datele la adresa 0x10012000 si afisati sirul sortat pe consola.

57. Scrieti un program care afiseaza primele n perechi de numere prime impare consecutive (n - numar impar citit de la tastatura). Exemplu: (3,5), (5,7), etc.

58. Scrieti un program, în limbaj de asamblare DLX, care citeste n numere întregi de la tastatura prin intermediul modulului Input.s (vezi lucrarea Investigatii Arhitecturale Utilizând Simulatorul DLX) si calculeaza maximul, minimul si suma numerelor si le depune succesiv în memoria DLX la adresa 0x1500.

59. **(*Conjectura lui Goldbach*)** Scrieti un program în limbaj de asamblare DLX, care citeste de la tastatura, prin intermediul modulului Input.s, un numar *n* par, n>6. Sa se determine toate reprezentarile lui *n* ca suma de numere prime, suma cu numar minim de termeni. Afisarea se face pe consola pe fiecare linie câte o solutie.

60. Proiectati automatul de control cache, într-un sistem multimicroprocesor simetric pe bus comun, având în vedere ca un bloc din cache se poate afla într-una din starile: PARTAJAT, EXCLUSIV sau INVALID.

61. Se considera un sistem multimicroprocesor (SMM) cu N microprocesoare legate printr-o retea de interconectare (RIC) la N module fizice de memorie. În ce ar consta si ce ar permite o RIC cu largime de banda maxima ? Dar una cu largime de banda minima ?

62. Într-un SMM cu memorie centrala partajata si în care fiecare procesor detine un cache propriu, un procesor initiaza o scriere cu MISS într-un anumit bloc aflat în starea “partajat” (“shared”). Precizati procesele succesive care au loc în urma acestei operatii.

63. Într-un SMM un procesor initiaza o citire cu MISS la un bloc invalid în cache-ul propriu, blocul aflându-se în copia exclusiva într-alt procesor. Precizati concret procesele succesive care au loc în urma acestei operatii.

64. Se considera un procesor cu un cache conectat la o memorie principala printr-o magistrala (bus de date) de 32 de biti; un acces cu hit în cache dureaza un ciclu de tact. La un acces cu miss în cache întregul bloc trebuie extras din memoria principala prin intermediul magistralei. O tranzactie pe bus consta dintr-un ciclu de tact pentru trimiterea adresei pe 32 de biti spre memorie, 4 cicli de tact în care are loc accesarea memoriei (strobarea datelor pe bus) si 1 ciclu pentru transferarea fiecarui cuvânt de date (32 octeti) în blocul din cache. Se presupune ca procesorul continua executia doar dupa ce ultimul cuvânt a fost adus în cache. Urmatorul tabel exprima rata medie de miss într-un cache de 1Mbyte pentru diverse dimensiuni a blocului de date.

Dimensiunea blocului (B), în cuvinte	**Rata de miss (m), în %**
1	4,5
4	2,4
8	1,6
16	1,0
32	0,75

Se cere:

a) Pentru care din blocuri (pentru ce dimensiune) se obtine cel mai bun timp mediu de acces la memorie ?

b) Daca accesul la magistrala adauga doi cicli suplimentari la timpul mediu de acces la memoria principala (disputa pentru ocuparea bus-ului), care din blocuri determina optima valoare pentru timpul mediu de acces la memorie ?

c) Daca latimea magistralei este dublata la 64 de biti, care este dimensiunea optima a blocului de date din punct de vedere al timpului mediu de acces la memorie ?

10. INDICATII DE SOLUTIONARE

1.

A. i1: LOAD R_1, 9(R_5)
i2: ADD R_6, R_1, R_3

a)

i1:	IF	ID	ALU	MEM	WB				
		IF	ID	ALU	MEM	WB			
			IF	ID	ALU	MEM	WB		
i2:				IF	ID	ALU	MEM	WB	

Delay = 2 cicli.

b) aplicând *forwarding*

i1:	IF	ID	ALU	MEM	WB			
		IF	ID	ALU	MEM	WB		
i2:			IF	ID	ALU	MEM	WB	

***Delay* = 1 ciclu.**

B. i1: ADD R_1, R_6, R_7
i2: LOAD R_5, 9(R_1)

a)

i1:	IF	ID	ALU	MEM	WB				
		IF	ID	ALU	MEM	WB			
			IF	ID	ALU	MEM	WB		
i2:				IF	ID	ALU	MEM	WB	

***Delay* = 2 cicli.**

b) aplicând forwarding

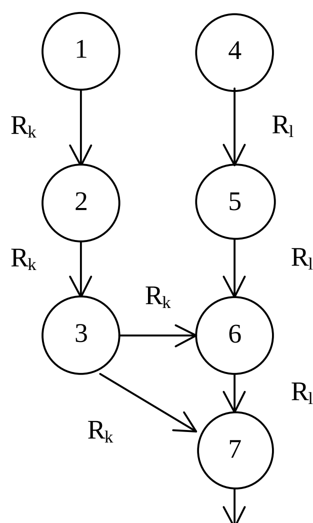
i1:	IF	ID	ALU	MEM	WB		
i2:		IF	ID	ALU	MEM	WB	

***Delay* = 0 cicli.** (necesar R_1 la începutul fazei ALU a i2 pentru calcularea adresei de memorie de unde va fi citita data; daca modul de adresare ar fi indirect registru, R_1 ar fi necesar doar la începutul fazei MEM, deci între cele doua instructiuni dependente RAW mai poate exista înca o a treia fara a implica penalizari)

2.

i1: SUB R_k, R_i, #4
i2: LOAD R_k, (R_k+0)
i3: ADD R_k, R_k, #12
i4: ADD R_l, #5, R_j
i5: LOAD R_l, (R_l+0)
i6: XOR R_l, R_l, R_k
i7: AND R_m, R_k, R_l

a)



b)

I1:	IF	ID	ALU / MEM	WB		
		IF	ID	ALU / MEM	WB	
I2:			IF	ID	ALU / MEM	WB

***Delay* = 1 ciclu.**

1 – nop – 2 – nop – 3 – 4 – nop – 5 – nop – 6 – nop – 7 ⇒ T_{ex} = 12 cicli executie.

c) 1 – 4 – 2 – 5 – 3 – nop – 6 – nop – 7 ⇒ T_{ex} = 9 cicli executie.

d) Aplicând forwarding-ul rezulta ***Delay* = 0 cicli** ⇒ Secventa se executa:

1 – 4 – 2 – 5 – 3 – 6 – 7 ⇒ T_{ex} = 7 cicli executie.

3.
a) **IRmax** = 2 instr. / ciclu.

FR = 4 instr. / ciclu ⇒ fiind 20 instructiuni de executat ⇒ vor fi 5 accese (citiri) la IC.
RmissIC = 40% ⇒ 2 accese la IC vor fi cu miss. Consideram primele 2 accese la cache.

Configuratia buffer-ului de prefetch în fiecare ciclu de executie:

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
					i16							
			i12		i15			i20				
	i8		i11	i12	i14	i16		i19				
	i7	i8	i10	i11	i13	i15	i16	i18	i20			
i4	i6	i7	i9	i10	i12	i14	i15	i17	i19			
i3	i5	i6	i8	i9	i11	i13	i14	i16	i18	i20		
i2	i4	i5	i7	i8	i10	i12	i13	i15	i17	i19		
i1	i3	i4	i6	i7	i9	i11	i12	i14	i16	i18	i20	

C1 – se executa în 10 cicli (primul miss în IC – aducere instructiuni din memoria principala); în buffer nefiind nici o instructiune nu se executa nimic.
C2 – se executa în 10 cicli (al doilea miss în IC – aducere instructiuni din memoria principala); în buffer sunt instructiuni, se executa i1 si i2.

C3 – se executa în 2 cicli (nu se face fetch instructiune nefiind spatiu disponibil în buffer); în buffer sunt instructiuni, se executa i3 datorita dependentelor RAW dintre i3 si i4.
C4 – se executa în 2 cicli (hit în IC – aducere instructiuni din cache); se executa i4 si i5.
C5 – se executa în 2 cicli (nu se face fetch instructiune nefiind spatiu disponibil în buffer); se executa i6 datorita dependentelor RAW dintre i6 si i7.
C6 – se executa în 2 cicli (hit în IC – aducere instructiuni din cache); se executa i7 si i8.
C7 – se executa în 2 cicli (nu se face fetch instructiune); se executa i9 si i10.
C8 – se executa în 2 cicli (nu se face fetch instructiune); se executa i11 datorita dependentelor RAW dintre i11 si i12.
C9 – se executa în 2 cicli (hit în IC – aducere instructiuni din cache); se executa i12 si i13.
C10 – se executa în 2 cicli (nu se mai face fetch instructiune – s-au adus toate instructiunile); se executa i14 si i15.
C11 – se executa în 2 cicli; se executa i16 si i17.
C12 – se executa în 2 cicli; se executa i18 si i19.
C13 – se executa în 2 cicli; se executa i20 – s-a golit buffer-ul nu mai sunt instructiuni de executat.

T_{ex} = 10 (C1) + 10 (C2) + 2 (C3) + 2 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2 (C8) + 2 (C9) + 2 (C10) + 2 (C11) + 2 (C12) + 2 (C13) = 42 impulsuri de tact

IR = 20 instr. / 42 imp. = 0,476 instr. / tact

b) **IRmax** = 4 instr. / ciclu.
Configuratia buffer-ului de prefetch în fiecare ciclu de executie:

C1	C2	C3	C4	C5	C6	C7	C8
		i12	i16				
	i8	i11	i15	i16	i20		
i4	i7	i10	i14	i15	i19		
i3	i6	i9	i13	i14	i18	i20	
i2	i5	i8	i12	i13	i17	i19	
i1	i4	i7	i11	i12	i16	i18	

T_{ex} = 10 (C1) + 10 (C2) + 2 (C3) + 2 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2 (C8) = 32 impulsuri de tact

IR = 20 instr. / 32 imp. = 0,625 instr. / tact

Cresterea performantei = (IR(4) – IR(2)) / IR(2) = 31%

4.

```
a) MOV     R10, R6
   ST      (R9), R6

b) ADD     R3, R5, #4    /* Calculeaza adresa pentru ST */
   ADD     R4, R6, #8    /* Calculeaza adresa pentru LD */
   LD      R9, 8(R6)     /* Daca adresele difera efectuam anticipat LD */
   BEQ     R3, R4, et    /* Daca adresele sunt egale, salt pentru a încarca
                            în R9 valoarea corecta (R8)*/
   J       end
et:   MOV  R9, R8
end:  ST 4(R5), R8       /* Daca adresele coincid sau nu se memoreaza R8
la adresa ceruta */
```

5.

a)	MOV R6, R7 ADD R3, R6, R5	Ra)	MOV R6, R7 ADD R3, R7, R5
b)	MOV R6, #4 ADD R7, R10, R6 LD R9, (R7)	Rb)	MOV R6, #4 ADD R7, R10, #4 LD R9, 4(R10)
c)	MOV R6, #0 ST 9(R1), R6	Rc)	MOV R6, #0 ST 9(R1), R0
d)	MOV R5, #4 BNE R5, R3, Label	Rd)	MOV R5, #4 BNE R5, #4, Label
e)	ADD R3, R4, R5 MOV R6, R3	Re)	ADD R3, R4, R5 ADD R6, R4, R5

6.
a) IRmax = 4 instr. / ciclu.

FR = 4 instr. / ciclu ⇒ fiind 32 instructiuni de executat ⇒ vor fi 8 accese (citiri) la IC.
$RmissIC_1$ = 50% ⇒ 4 accese la IC vor fi cu miss. Consideram primele 4 accese la cache.
Configuratia buffer-ului de prefetch în fiecare ciclu de executie:

C1 (F1)	C2 (F2)	C3 (F3)	C4 (F4)	C5	C6	C7	C8	C9	C10	C11	C12	C13
						i24	i28					
		i12	i16			i23	i27	i28	i32			
	i8	i11	i15	i16	i20	i22	i26	i27	i31	i32		
i4	i7	i10	i14	i15	i19	i21	i25	i26	i30	i31		
i3	i6	i9	i13	i14	i18	i20	i24	i25	i29	i30		
i2	i5	i8	i12	i13	i17	i19	i23	i24	i28	i29		
i1	i4	i7	i11	i12	i16	i18	i22	i23	i27	i28	i32	

C1 – se executa în 10 cicli (primul miss în IC – aducere instructiuni din memoria principala); în buffer nefiind nici o instructiune nu se executa nimic.
C2 – se executa în 10 cicli (al doilea miss în IC – aducere instructiuni din memoria principala); în buffer sunt instructiuni, se executa i1, i2 si i3.
C3 – se executa în 10 cicli (al treilea miss în IC – aducere instructiuni din memoria principala); în buffer sunt instructiuni, se executa i4, i5 si i6 datorita dependentelor RAW dintre i6 si i7.
C4 – se executa în 10 cicli (al patrulea miss în IC – aducere instructiuni din memoria principala); se executa i7, i8, i9 si i10.
C5 – se executa în 2 cicli (nu se face fetch instructiune nefiind spatiu disponibil în buffer); se executa i11 datorita dependentelor RAW dintre i11 si i12.
C6 – se executa în 2 cicli (hit în IC – aducere instructiuni din cache); se executa i12, i13, i14 si i15.
C7 – se executa în 2 cicli (hit în IC); se executa i16 si i17 datorita dependentelor RAW dintre i17 si i18.
C8 – se executa în 2 cicli (hit în IC); se executa i18, i19, i20 si i21.
C9 – se executa în 2 cicli (nu se face fetch instructiune nefiind spatiu disponibil în buffer); se executa i22 datorita dependentelor RAW dintre i22 si i23.
C10 – se executa în 2 cicli (hit în IC); se executa i23, i24, i25 si i26.

C11 – se executa în 2 cicli; (nu se mai face fetch instructiune – s-au adus toate instructiunile); se executa i27 datorita dependentelor RAW dintre i27 si i28.
C12 – se executa în 2 cicli; se executa i28, i29, i30 si i31.
C13 – se executa în 2 cicli; se executa i32 – s-a golit buffer-ul nu mai sunt instructiuni de executat.

T_{ex} = 10 (C1) + 10 (C2) + 10 (C3) + 10 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2 (C8) + 2 (C9) + 2 (C10) + 2 (C11) + 2 (C12) + 2 (C13) = 58 impulsuri de tact

IR = 32 instr. / 58 imp. = 0,55 instr. / tact

b) **FR** = 8 instr. / ciclu ⇒ fiind 32 instructiuni de executat ⇒ vor fi 4 accese (citiri) la IC.
RmissIC$_2$ = 50% × **RmissIC**$_1$ ⇒ **RmissIC**$_2$ = 25% ⇒ 1 acces la IC va fi cu miss. Consideram primul acces la cache.

Configuratia buffer-ului de prefetch în fiecare ciclu de executie:

C1 (F1)	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
i8			i16			i24				i32			
i7			i15			i23	i24			i31			
i6			i14			i22	i23			i30			
i5	i8		i13	i16		i21	i22			i29	i32		
i4	i7		i12	i15		i20	i21			i28	i31		
i3	i6		i11	i14		i19	i20	i24		i27	i30		
i2	i5	i8	i10	i13		i18	i19	i23	i24	i26	i29		
i1	i4	i7	i9	i12	i16	i17	i18	i22	i23	i25	i28	i32	

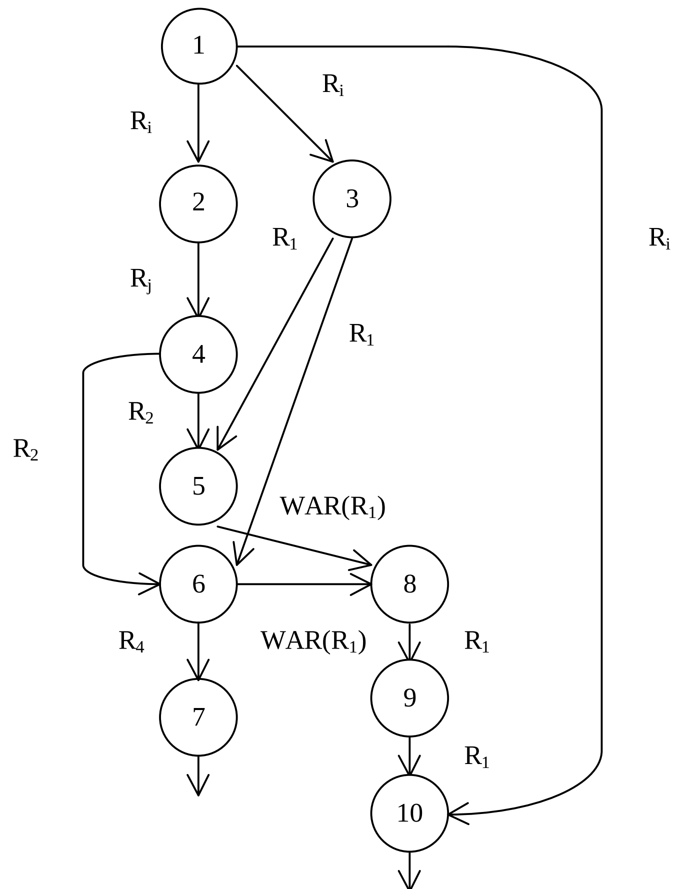
T_{ex} = 10 (C1) + 2 (C2) + 2 (C3) + 2 (C4) + 2 (C5) + 2 (C6) + 2 (C7) + 2 (C8) + 2 (C9) + 2 (C10) + 2 (C11) + 2 (C12) + 2 (C13) + 2 (C14) = 36 impulsuri de tact

IR = 32 instr. / 36 imp. = 0,89 instr. / tact

Cresterea performantei = (IR(FR=8) – IR(FR=4)) / IR(FR=4) = 61,8% (substantiala).
Limitarea prformantei consta în capacitatea buffer-ului de prefetch egala cu rata de fetch (**IBS=FR**).

7.

a)



I1:	IF	ID	ALU / MEM	WB		
		IF	ID	ALU / MEM	WB	
I2:			IF	ID	ALU / MEM	WB

***Delay* = 1 ciclu.**

1 – nop – 2 – 3 – 4 – nop – 5 – 6 – nop – 7 – 8 – nop – 9 – nop – 10 ⇒ T_{ex} = 15 cicli executie.

b) Dupa aplicarea tehnicii de *renaming* la registrul R_1 în instructiunea i8 obtinem:

i8: ADD R_1', R_{13}, R_{14}
i9: DIV R_1', R_1', #2
i10: STORE (R_i), R_1'

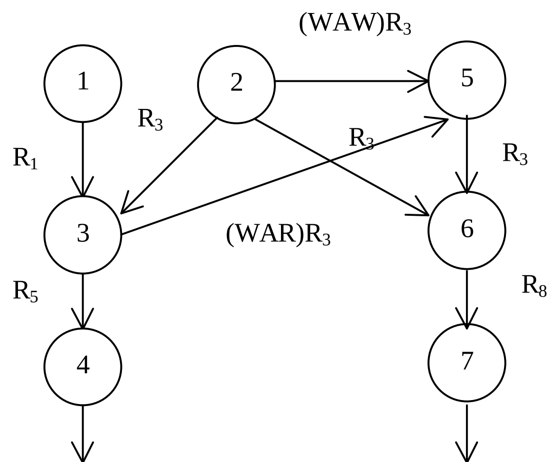
1 – 8 – 2 – 3 – 4 – 9 – 5 – 6 – 10 – 7 ⇒ T_{ex} = 10 cicli executie.

c) Prin înlocuirea registrilor R_{13} cu R_3 si R_{14} cu R_4, secventa realizeaza determinarea maximului dintre elementele x[i] si x[i+4] ale unui sir de date stocat în memorie. Se obtine formula:

$$\max(x[i], x[i+4]) = \frac{(x[i+4] + x[i]) + |x[i+4] - x[i]|}{2}$$

8.

a)



Secventa se executa astfel:
1 – 2 – nop – 3 – nop – 4 – 5 – nop – 6 – nop – 7 ⇒ T_{ex} = 11 cicli executie.

b) Se aplica tehnica de *renaming* pe ramura 5 – 6 – 7 ⇒ R_3 = R_3'.
Astfel având 3 unitati ALU la dispozitie executia secventei ar fi urmatoarea:

Nr. Tact	Instructiuni executate în paralel
	1 – 2 – 5
	3 – 6
	4 – 7

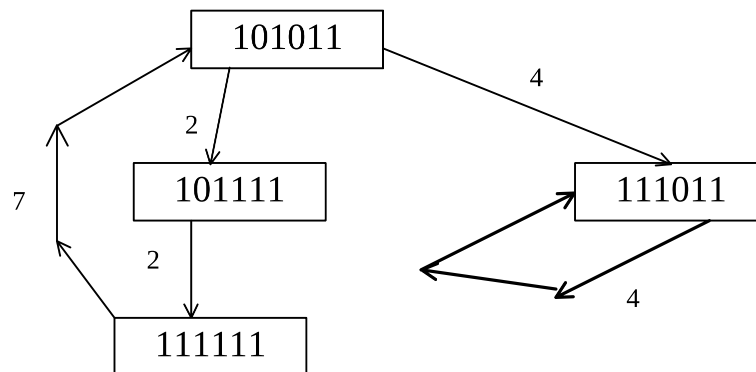
T_{ex} = 3 cicli executie.

9.

a) Da. La startarea sistemului se încarca programul *încarcator* care verifica perifericele, citeste informatii din EPROM. E posibil sa existe hit la citire din cache care nefiind initializat contine "*prostii*" si daca nu ar exista bitul de validare (V) sistemul ar prelua valoarea din cache eronata.

b) O exceptie *Page Fault* implica întotdeauna o exceptie TLB miss. Reciproc, nu e obligatoriu.

11.



Daca alegem strategia Greedy obtinem rata de procesare $IR = \frac{3}{11} = 0{,}27\ {}^{instr}\!/_{ciclu}$

Daca alegem strategia non - Greedy rata de procesare obtinuta este: $IR = \frac{1}{4} = 0{,}25\ {}^{instr}\!/_{ciclu}$

În acest caz e mai avantajos sa alegem stategia Greedy.

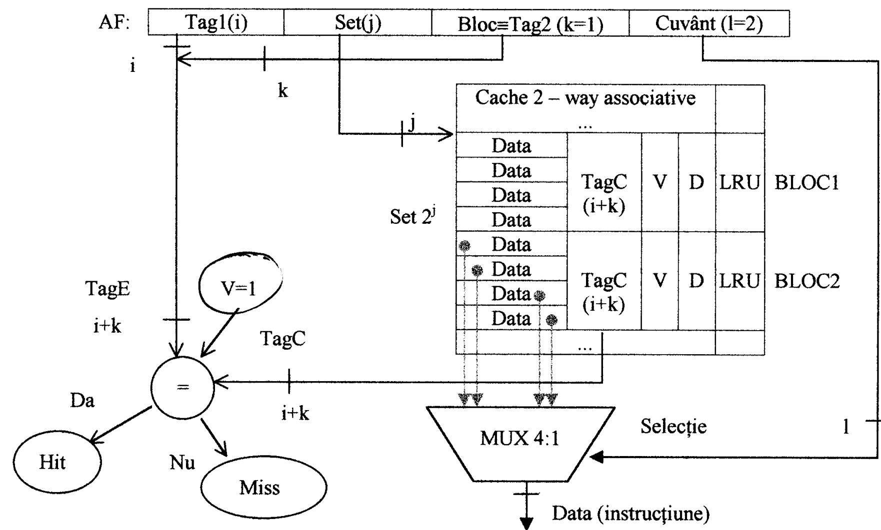
21.

a.

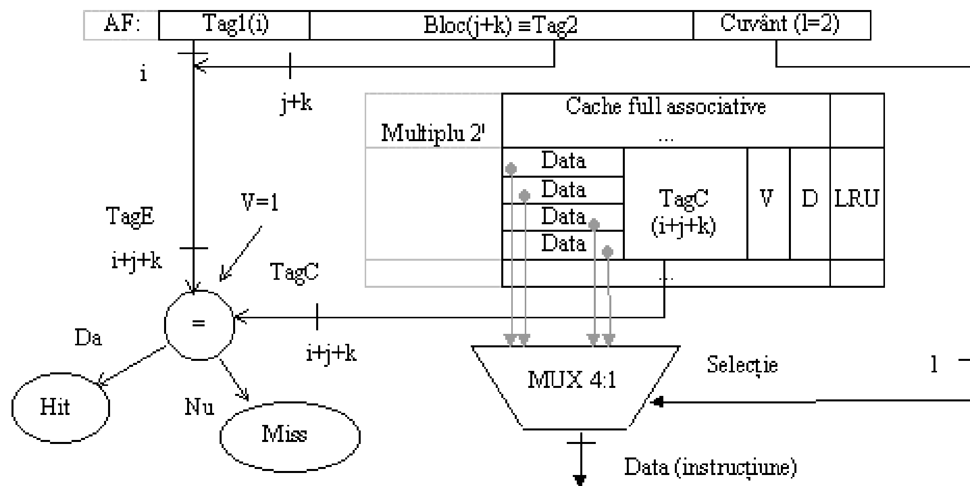
Cache-ul **semiasociativ** contine 2^j seturi, fiecare set contine 2 blocuri.

V – bit de validare (0 – nu e valida data; 1 – valida;). Initial are valoarea 0. Este necesar numai pentru programe automodificabile la cache-urile de instructiuni.

D – bit Dirty. Este necesar la scrierea în cache-ul de date.



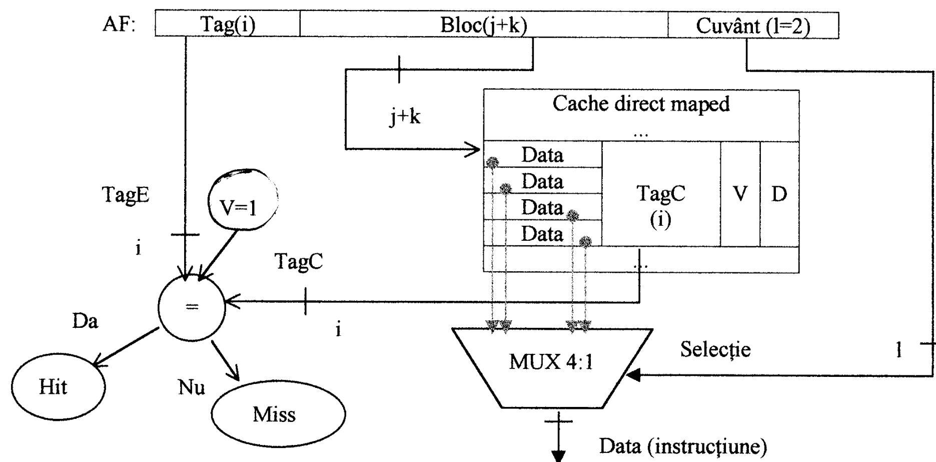
b.



În cazul cache-urilor **full asociative** dispare câmpul set. Practic exista un singur set. Datele pot fi memorate oriunde (în orice bloc) în cache.

c.

În cazul cache-urilor **mapate direct**, datele vor fi memorate în acelasi loc de fiecare data când sunt accesate. Din acest motiv vom sti la fiecare acces ce data va fi evacuata din cache, nemaifiind necesar câmpul LRU (evacuare implicita).



22. a. Procesarea «Out of Order» a instructiunilor cu referire la memorie într-un program este dificila datorita accesarii aceleiasi adrese de memorie de catre o instructiune Load respectiv una Store. Exemplificam pe secventa de instructiuni urmatoare:

Presupunem ca la adresa 2000h avem memorata valoarea 100.

```
LOAD   R1, 2000h
ADD         R1, R1, #12
STORE  R1, 2000h
```

În urma executiei în registrul R1 si implicit la adresa 2000h vom avea valoarea 112.
Prin *Out of Order* aplicat instructiunilor cu referire la memorie valoarea din registrul R1 precum si cea din memorie de la adresa 2000h ar fi alterata.

b. Secventele de program în limbajul unui procesor RISC ar deveni:

R1 ← x[i];
(R1) → a[2i]; STORE Adr1
R2 ← a[i+1]; LOAD Adr2
R6 ← (R2) + 5;
(R6) → y[i];

Cele doua instructiuni cu referire la memorie ar fi paralelizabile (executabile Out of Order) daca i ≠ 1 (Adr1 ≠ Adr2).

Benchmark-ul **B2** s-ar procesa mai repede pe un procesor superscalar cu executie *Out of Order* a instructiunilor, pentru ca cele doua aliasuri (i=1) au fost scoase în afara buclei, prin urmare în cadrul buclei **B2**, paralelizarea Load / Store e posibila.

23. a. Vezi pr. 44 a), b).
Prin «renaming» aplicat registrului R1 putem elimina hazardul WAW dintre instructiunile (1 si 5) si (2 si 6), deci le putem trimite în executie în acelasi ciclu.

Executia: tact 1 - instructiunile: 1, 5, 3.
tact 2 - instructiunile: 2, 6.
tact 3 - instructiunea: 4.

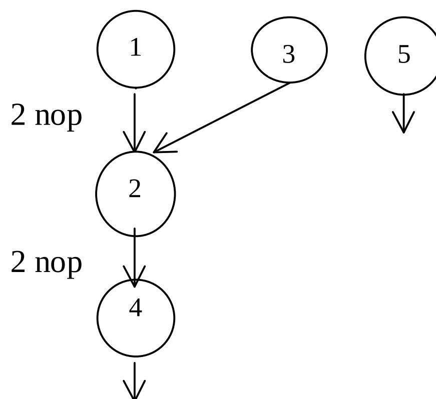
24. Nivelul **WR** este mai prioritar datorita hazardurilor RAW între doua instructiuni succesive (vezi si **50.b**).

25. $FR_{med} = 5$; Da. Predictia a N PC-uri simultan implica $FR_{med} \cong N \times 5$.

26. C $\rightarrow$ (N-1) + (N-2) + ... + 2 + 1 = N(N-1) / 2 comparatoare RAW.
Pentru (N+1) instructiuni => N(N+1) / 2 comparatoare $\rightarrow$ C'
C' / C = N(N+1) / N(N-1) => C' = C(N+1) / (N-1)

27. a) A. b) F. c) F. d) F.

28. a)



b) 5 cicli pentru instr. + 4 cicli nop = 9 cicli executie
c) 1 – 3 – 5 – 2 – nop – nop – 4 => 7 cicli executie

29. a.

Instrucțiunea de salt:	IF	ID	ALU	MEM	WB			
		IF	ID	ALU	MEM	WB		
			IF	ID	ALU	MEM	WB	
				IF	ID	ALU	MEM	WB

Branch delay slot = 2 cicli.

Motivul implementarii de busuri si memorii cache separate pe instructiuni, respectiv date în cazul majoritatii procesoarelor RISC (pipeline) consta în faptul ca nu exista coliziuni la memorie de tipul (IF, MEM).
Instructiunile CALL / RET sunt mari consumatoare de timp în cazul procesoarelor CISC datorita salvarilor si restaurarilor de registri (registrul stare program, registrul de flaguri, PC) pe care acestea le executa de fiecare data când sunt apelate. Evitarea consumului de timp în cazul microprocesoarelor RISC se face prin implementarea *ferestrelor de registre* sau prin inlining-ul procedurilor (utilizarea de macroinstructiuni).

30. Codificarea instructiunii este urmatoarea:

7..........0
Opcode
Deplasament (1)
Deplasament (2)

Se efectueaza urmatoarele operatii:

Operatia executata	**Durata executie (cicli)**
Fetch Instructiune	6
Fetch Deplasament (1)	4
Fetch Deplasament (2)	4
Calcul adresa de memorare	2
Scriere A în memorie	4

Total cicli executie = 20.

31. a) **Fals**! În caz de hit, pe baza compararii tag-urilor, se citeste si data respectiva => acces simultan la tag si date.
b) **Fals**! Fiind hit tag-ul nu mai are sens sa-si modifice valoarea.
c) **Fals**! Datorita interferentelor alternative, rata de hit scade.

32. i5: LOAD R_l, ($\underline{R_l}$+0)

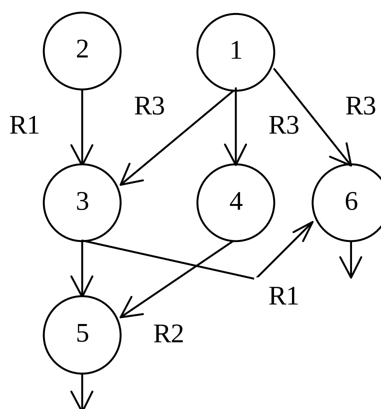
33. Nu! O regenerare transparenta (dureaza 250ns) trebuie sa “încapa” între 2 accese la DRAM ale microprocesorului. Cum un ciclu extern al procesorului dureaza 3 tacte (150ns) regenerarea transparenta este imposibila la frecventa maxima a procesorului.

a. ALU: (R9) + 06; MEM: scriere R5 la adresa (R9) + 06;WB: nimic

b. ALU: (R8) + F3; MEM: citire de la adresa (R8) + F3;WB: scriere în R7;

c. ALU: (R7) + (R8); MEM: nimic; WB: scriere în R5.

35.
a. 1 - 2 - nop - 3 - 4 - nop - 5 - 6 => 8 cicli.



b. 1 - 2 - 4 - 3 - 6 - 5 => 6 cicli.

36.
a. Nr.tacte /s consumate pentru interogare mouse: 30×100 = 3000 tacte/ s

$$f = \frac{3000}{50\times10^{6}} = 0{,}006\%$$

b. $$\frac{\text{Nr.interogari}}{\text{s}} = \frac{50\,\frac{\text{ko}}{\text{s}}}{2\,\frac{\text{o}}{\text{acces interogare}}} = 25\times2^{10}\,\frac{\text{acces interogare}}{\text{s}}$$

Nr. tacte necesar pentru Nr.interogari/ s = $25\times2^{10}\times100$ tacte

$$f = \frac{25{,}6 \times 10^5}{50 \times 10^6} = 5{,}12\%$$

c. $$\frac{\text{Nr.interogari}}{\text{s}} = \frac{2\frac{\text{Mo}}{\text{s}}}{4\frac{\text{o}}{\text{acces interogare}}} = 0{,}5 \times 2^{20}\,\frac{\text{acces interogare}}{\text{s}}$$

Nr. tacte necesar pentru Nr.interogari/ s = $0{,}5\times2^{20}\times100$ tacte

$$f = \frac{0{,}5 \times 2^{20} \times 100}{50 \times 10^6} > 100\%$$

În cazul hard-disc-ului este imposibila comunicatia dintre procesor si periferic prin interogare. (Într-o secunda procesorul realizeaza 50×10^6 tacte, iar pentru un transfer cu o rata de 2 Mo/ s sunt necesare într-o secunda 50×2^{20} tacte, imposibil).

37.

a. AF = 20000h;
b. AL se memoreaza la adresa: E0BF2h (adresa para)
   AH se memoreaza la adresa: E0BF3h (adresa impara) [scriere pe cuvânt la adresa impara]
   SS = 2000h constituie informatie redundanta.

38.

a. AF = B0000h;
b. AL se memoreaza la adresa: 1FF1Ch (adresa para - [SS:(SP-1)])
   AH se memoreaza la adresa: 1FF1Dh (adresa impara) [scriere pe cuvânt la adresa impara [SS:(SP-2)]]

DS = 1F20h constituie informatie redundanta.

39. Nu! Daca se activeaza CREF si microprocesorul vrea sa acceseze apoi memoria, acesta trebuie pus în stare «WAIT». La activarea CREF în conditiile în care microprocesorul nu lucreaza cu memoria, regenerarea va astepta ca microprocesorul sa "atace" memoria, pentru ca dupa aceea sa se "strecoare".

40. a – cazul memoriei **mapate direct** cu 4 locatii.
Se acceseaza pe rând locatiile:

Locația accesată	0	8	0	6	8	10	8
	Tag	Tag	Tag	Tag	Tag	Tag	Tag
0	0(miss)	2(miss)	0(miss)	0	2(miss)	2	2(hit)
1	X	X	X	X	X	X	X
2	X	X	X	1(miss)	1	2(miss)	2
3	X	X	X	X	X	X	X

Rezulta în cazul memoriei mapate direct un singur acces cu hit $R_{miss} = 6 / 7 = 85.71\%$

b – cazul memoriilor semiasociative cu 2 seturi a câte 2 cuvinte.
Întrucât toate adresele sunt pare se acceseaza doar blocurile din setul 0**.**

Locația accesată	Setul 0		
0	0	X	Miss
8	0	8	Miss
0	0	8	Hit
6	0	6	Miss. Se evacuează 8.
8	8	6	Miss. Se evacuează 0.
10	8	10	Miss. Se evacuează 6.
8	8	10	Hit.

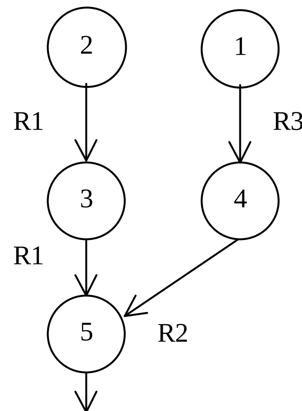
În cazul memoriei two-way asociative sunt 2 accese cu hit $R_{miss} = 5 / 7 = 71.42\%$

c – cazul memoriilor complet asociative.

Locația accesată	0	8	0	6	8	10	8
	Tag	Tag	Tag	Tag	Tag	Tag	Tag
0	0(miss)	0	0(hit)	0	0	0	0
1	X	8(miss)	8	8	8(hit)	8	8(hit)
2	X	X	X	6(miss)	6	6	6
3	X	X	X	X	X	10(miss)	10

În cazul memoriei complet asociative sunt 3 accese cu hit $R_{miss} = 4 / 7 = 57.14\%$

41. a. 1 - 2 - nop - 3 - 4 - nop - 5 => 7 cicli.



b. 1 - 2 - 4 - 3 - nop - 5 => 6 cicli.

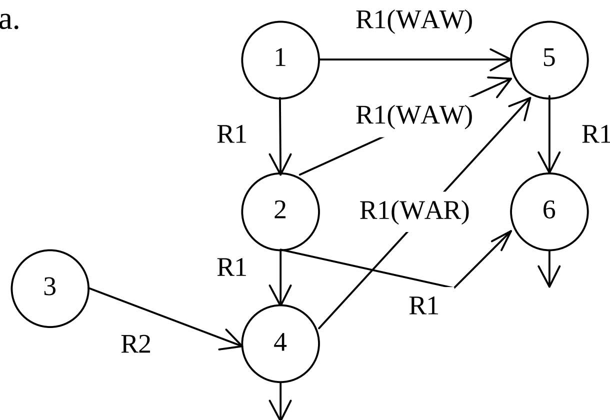
42. În 1 s se transfera 25×10^4 biti.
În x s se transfera 1 octet.

Rezulta $x = 8 / (25\times10^4) = 32\ \mu s$
Fie t_r = timpul disponibil dintre 2 transferuri succesive de octeti.
t_1 = timpul scurs între aparitia întreruperii si intrarea în rutina de tratare;
$t_1 = 2\ \mu s$;
t_2 = timpul în care este tratata rutina de întrerupere; $t_2 = 10\mu s$;
$t_r = x - t_1 - t_2 = (32 - 2 - 10)\ \mu s \Rightarrow t_r = 20\ \mu s$.

43. a. $\frac{(1-0,11)\times8,5+0,11\times6\times3}{8,5}=1,1229$ => diminuare a performantei cu ≈ 13%.

b. **Avantajul** implementarii unei pagini de capacitate «mare» într-un sistem de memorie virtuala consta în principiul localizarii acceselor, determinând o optimizare a acceselor la disc (se reduce numarul acestora).
Dezavantajul îl reprezinta aducerea inutila de informatie de pe disc (pierdere de timp), care trebuie apoi evacuata în cazul unei erori de tipul *PageFault*. Dimensiunea paginii se alege în urma unor simulari laborioase.

44. a.



1 - nop - 2 - 3 - nop - 4 - 5 - nop - 6 => 9 cicli executie.

b.

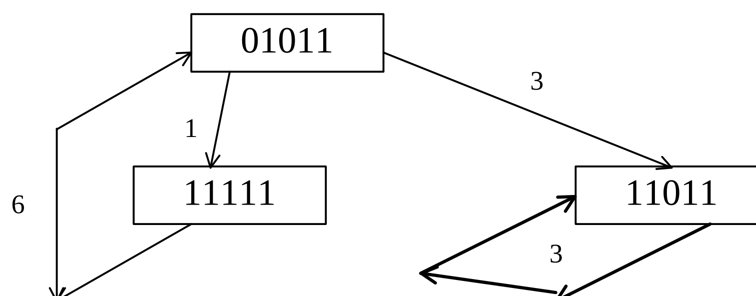
1: R1 ← (R11) + (R12)
5: R1’ ← (R14) + (R15)
3: R2 ← (R3) + 4
2: R1 ← (R1) + (R13)
6: R1’ ← (R1’) + (R16)
4: R2 ← (R1) + (R2)

1 - 5 - 3 - 2 - 6 - 4 => 6 cicli executie

45. Daca alegem strategia Greedy obtinem rata de procesare $IR = \frac{2}{7} \text{instr}/\text{ciclu}$

Daca alegem strategia non - Greedy rata de procesare obtinuta este: $IR = \frac{1}{3} \text{instr}/\text{ciclu}$

În acest caz e mai avantajos sa alegem stategia non - Greedy.



46. Algoritmul lui Tomasulo permite anularea hazardurilor WAR si WAW printr-un mecanism hardware de redenumire a registrilor, favorizând executia multipla si *Out of Order* a instructiunilor. Mecanismul de «forwarding» implementat prin arhitectura Tomasulo (statii de rezervare) determina reducerea semnificativa a presiunii la «citire» asupra setului de registri logici, înlaturând o mare parte din dependentele RAW dintre instructiuni [33].
În cazul unei arhitecturi TTA, numarul de registri generali poate fi redus semnificativ datorita faptului ca trebuie stocate mai putine date temporare, acestea circulând direct între unitatile de executie (FU - unitati functionale), nemaifiind necesara memorarea lor în registri. «Forwarding-ul» datelor este realizat software prin program, spre deosebire de procesoarele superscalare care realizeaza acest proces prin hardware folosind algoritmul lui Tomasulo [33].

47. O instructiune de tip RETURN este dificil de predictionat printr-un predictor hardware datorita **fenomenului de interferenta a salturilor** . Acesta apare în cazul unei predictii incorecte datorate exclusiv adresei de salt incorecte din tabela de predictii, care a fost modificata de catre un alt salt anterior. Instructiunile de tip RETURN reprezinta salturi care-si modifica dinamic adresa tinta, favorizând dese aparitii ale fenomenului de interferenta. Analog se întâmpla si în cazul salturilor în mod de adresare indirect [33].
Noutatea «principiala» a predictoarelor corelate pe doua nivele consta în faptul ca predictia unei instructiuni tine cont de predictia ultimelor *n* instructiuni de salt anterioare; se foloseste un registru de predictie (registru binar de deplasare) care memoreaza istoria ultimelor *n* instructiuni de salt. Valoarea acestui registru concatenata cu cei mai putini semnificativi biti ai PC-ului instructiunii de salt curente realizeaza adresarea cuvântului de predictie din tabela de predictie [33, 45].

50. a. Sumatorul “sum 2” este activat de o instructiune de branch, pentru calculul adresei de salt.
b. Nivelul **WR** este mai prioritar datorita hazardurilor RAW. Operatia de citire ar putea avea nevoie de un registru în care nu s-a înscris înca rezultatul final, rezulta prioritatea scrierii fata de citire.
c. În cazul unei instructiuni de tip LOAD, unitatea ALU are rol de calcul adresa.
d. În latch-ul EX/MEM se memoreaza valoarea (R7+05).

52. Se citesc caractere de la intrare pâna se întâlneste conditia de iesire, si se salveaza acestea în stiva. Conditia de iesire o constituie tastarea caracterului ‘0’. La întâlnirea sa se afiseaza caracterul curent (‘0’ – primul caracter afisat) si se va apela functia de afisare. În cadrul acestei functii vom prelua din stiva caracterele memorate si le vom tipari.

Obs. E necesar un parametru pentru contorizarea caracterelor scrise în stiva.

Programul modificat pentru a nu afisa si caracterul ‘0’, difera prin faptul ca la întâlnirea conditiei de iesire se apeleaza direct functia de afisare.

53. Este esential sa calculam cmmdc (cel mai mare divizor comun) dintre cele doua numere, cmmmc (cel mai mic multiplu comun) putându-se calcula apoi din formula:

$$cmmdc(a,b) \times cmmmc(a,b) = a \times b$$

Pentru calcularea cmmdc(a,b) folosim recursivitatea:

$$Cmmdc(a,b) = \begin{cases} a, & \text{daca } a = b \\ Cmmdc(a,b-a), & \text{daca } a < b \\ Cmmdc(b,a-b), & \text{daca } a > b \end{cases}$$

Se apeleaza recursiv functia având ca noi parametri minimul dintre cele doua numere si modulul diferentei dintre cele doua valori, pâna la întâlnirea conditiei de iesire (a = b), în a (si b) aflându-se chiar cel mai mare divizor comun.

54. Semnificatia celor 3 tije este urmatoarea:

A – sursa
B – destinatie
C – manevra

Problema pentru *n* discuri se rezolva usor daca putem rezolva pentru (n-1) discuri, deoarece rezolvând-o pe aceasta, vom putea muta primele (n-1) discuri de pe tija A(sursa) pe C(manevra), apoi discul *n* (cu diametrul cel mai mare) de pe tija A(sursa) pe tija B(destinatie) si din nou cele (n-1) discuri de pe tija C(manevra) pe tija B(destinatie).

Conditia de iesire din subrutina o constituie problema transferarii unui singur disc (n=1) de pe sursa pe destinatie. Pasii algoritmului sunt:

Se citesc de la tastatura numarul de discuri (*n*), si identificatorii (caractere) tijelor sursa, destinatie si manevra si se apeleaza subrutina *hanoi* având ca parametrii efectivi cele patru valori citite anterior. În cadrul functiei *hanoi* se executa:

Se salveaza în stiva adresa de revenire si cadrul de stiva. Se testeaza daca se îndeplineste conditia de iesire din subrutina.

Daca **da**, se afiseaza transferul (tija sursa si tija destinatie), se reface continutul registrilor PC si fp din stiva si se actualizeaza SP. Se executa instructiunea de la adresa data de PC.

Daca **nu**, se executa secventa:

a. Se salveaza în stiva registrii corespunzatori parametrilor (tijelor).
b. Se actualizeaza numarul de discuri, n ← (n-1) si rolul fiecarei tije (noua destinatie va fi tija C, tija A va fi sursa iar tija B va fi manevra). Se reapeleaza *hanoi*. [Se muta cele (n-1) discuri de pe tija sursa pe tija de manevra].
c. Se refac parametrii din stiva (tijele). Se afiseaza transferul [cel de-al *n* - lea disc de pe tija sursa(A) pe tija destinatie(B)].
d. Se salveaza în stiva registrii corespunzatori parametrilor (tijelor).
e. Se actualizeaza numarul de discuri, n ← (n-1) si rolul fiecarei tije (noua destinatie va fi tija B, tija C este sursa iar tija A va fi manevra). Se reapeleaza *hanoi*. [Se muta cele (n-1) discuri de pe tija de manevra pe cea destinatie].

55. Se modifica programul de calcul al factorialului unui numar, prezentat în limbaj de asamblare MIPS (vezi lucrarea "*Investigatii Arhitecturale Utilizând Simulatorul SPIM*"), calculând în paralel cu factorialul si aranjamentele, folosind formulele:

$$C_n^k = \frac{n!}{(n-k)!k!}; \qquad A_n^k = \frac{n!}{(n-k)!}$$

Parametrii *n* si *k* se citesc de la tastatura si sunt salvati în stiva. Se intra în subrutina unde se executa:

Se verifica daca k = 1 (conditia de iesire din subrutina).

Daca **da**, se seteaza în registrii rezultat valoarea 1, punctul de plecare în calcularea produselor 1×2× ... ×k si n×(n-1)× ... ×[n-(k-1)]. Se reface continutul registrilor PC si fp din stiva si se actualizeaza SP. Se executa instructiunea de la adresa data de PC.

Daca **nu**, se executa secventa:

a. Se actualizeaza parametrii n ← (n-1) si k ← (k-1). Se reapeleaza subrutina.
b. Se preiau din stiva termenii salvati si se înmultesc cu rezultatele calculate pâna la acest pas.
c. Se reface continutul registrilor PC si fp din stiva si se actualizeaza SP. Se executa instructiunea de la adresa data de PC.

În încheierea programului se calculeaza combinarile cu formula raportului dintre aranjamente si permutari si afiseaza valorile aranjamentelor, permutarilor si combinarilor.

56. Se citeste dimensiunea sirului, si elementele care se vor memora la adresa ceruta. Este necesara o variabila booleana, initial având valoarea 1 (true), care specifica daca au fost facute interschimbari între elementele sirului. Se apeleaza rutina *bubble* în care au loc urmatoarele:

Variabila booleana este resetata (ia valoarea 0).

Se porneste de la primul element al sirului si se compara elementele învecinate. Daca doua dintre acestea nu îndeplinesc relatia de ordine ceruta (impusa), se interschimba (fiecare element se memoreaza la adresa celuilalt) si se seteaza variabila booleana. Se parcurge tot sirul, astfel ca dupa o prima parcurgere elementul maxim (sau minim) se va afla pe ultima pozitie din sir.

Daca a avut loc cel putin o interschimbare se reia algoritmul. Altfel, sirul se gaseste în memorie sortat si va fi afisat pe consola.

57. Se parcurge sirul numerelor naturale din doi în doi (ne intereseaza doar numerele impare) începând cu numarul 3 (primul numar impar prim) si se determina daca este prim sau nu (vezi lucrarea "*Investigatii Arhitecturale Utilizând Simulatorul SPIM*"). În cazul în care numarul este prim (fie acesta *k*) se verifica daca si urmatorul numar impar (*k*+2) este prim si daca **da** se afiseaza perechea de numere. Se testeaza daca am ajuns la numarul de perechi de numere prime cerut si daca nu se continua algoritmul cu numarul prim mai mic. În momentul în care un numar se dovedeste a nu fi prim se trece la urmatorul numar impar.

58. Se citesc prin intermediul modulului **Input.s** usor modificat, dimensiunea unui sir si elementele sale ca numere întregi pozitive. Se stocheaza în memoria DLX la adresa specificata. Se seteaza suma si elementul maxim pe 0, iar minimul la o valoare maxima admisa (fie 0x7fff). Se parcurge sirul si se executa operatiile:

a. Se însumeaza în registrul suma - (fie r15) – suma numerelor anterior citite cu cea a elementului curent.

b. Daca elementul curent (citit din memorie), este mai mare decât maximul, atunci maximul devine elementul curent.
c. Daca elementul curent este mai mic decât minimul, atunci minimul devine elementul curent.
d. Se reia punctul a. pâna am parcurs tot sirul.
e. Se afiseaza suma, maximul si minimul în fereastra DLX-IO.

61. Retea de tip crossbar, permite implementarea oricarei bijectii procesoare-module memorie; retea unibus, un singur procesor master la un moment dat.

62. 1. Operatie "*write miss*" pe busul comun.
2. Citire bloc de la unul din cache-urile care îl detine ("*snooping*").
3. Toate procesoarele care au detinut respectivul bloc în cache-uri, îl trec în starea "*invalid*" (V=0).
4. Procesorul care l-a citit în pasul 2, îl scrie si îl pune în cache în starea "exclusiv".

11. CE GASITI PE CD ?

Plecând de la premisa ca tehnica de calcul si tehnologia informatiei constituie si va constitui si în România un element important în derularea oricarui tip de activitate, CD-ul ce însoteste aceasta carte cuprinde o serie de aplicatii, documente si alte fisiere care încearca sa sprijine la un cost rezonabil necesarul zilnic de cunoastere al studentilor din anii terminali, masteranzi si doctoranzi din domeniul ingineriei calculatoarelor si domenii conexe; al inginerilor si cercetatorilor.

Cea mai mare parte a programelor si fisierelor existente pe acest CD sunt în limba româna. Materialul este organizat pe mai multe directoare, fiecare dintre ele corespunzând, pe cât posibil, unui domeniu distinct. Acele programe sau fisiere care nu au putut fi incluse în nici unul dintre domeniile prezentate mai jos sunt grupate sub denumirea comuna de **Diverse**.

În plus, daca doriti sa cititi aceasta carte în format electronic, va punem la dispozitie textul carti în format *.pdf*. Pentru a putea citi acest format aveti la dispozitie programul Adobe Acrobat Reader pe care trebuie sa-l instalati pe calculatorul dumneavoastra.

11.1. SIMULAREA UNOR ARHITECTURI CU PARALELISM LA NIVELUL INSTRUCTIUNII

Aceasta sectiune realizeaza o descriere sumara a simulatoarelor, (structura, implementare, resurse necesare, ghid de utilizare, platforma de executie), prezente pe suportul CD-ROM care însoteste aceasta carte, dar si pe Internet la adresa http://www.sibiu.ro/ing/vintan/simulatoare.html.

Evaluarea performantelor arhitecturilor de calcul se face prin simulare, fara simulare fiind foarte dificil, practic imposibil, de estimat. Metodele de investigare folosite sunt cunoscute sub denumirile de *execution* sau *trace driven simulation*.

Simulatoarele sunt dedicate arhitecturilor RISC scalare (MIPS) si superscalare (DLX, simulatoare de cache-uri, predictoare de branch-uri, simulatoare la nivel de executie a instructiunilor *in order* respectiv *out of order* într-o arhitectura superscalara parametrizabila tipica - HSA [Hatfield Superscalar Architecture], scheduler - optimizator de cod - pentru arhitectura HSA).

Arhitectura familiei de microprocesoare RISC MIPS R2000/3000 este aprofundata prin intermediul simulatorului software numit SPIM, conceput de catre James R. Larus. Arhitectura MIPS, pe lânga succesul sau comercial, constituie unul dintre cele mai inteligibile si elaborate arhitecturi RISC.

SPIM/SAL este o portare pe WIN32S a SPIM, simulator scris pentru uz didactic în Facultatea de Calculatoare a Universitatii din Wisconsin, SUA. WIN32S este o extensie pe 32 de biti a sistemului Windows 3.1. Interfata programabila cu aplicatia a WIN32S este un subset al WIN32, un API suportat de sistemul de operare Windows NT. Aplicatii ce folosesc WIN32S API, cum sunt SPIM/SAL, sunt compilate într-un format executabil numit *executabil portabil* (PE) care ruleaza fie pe Microsoft Windows 3.1. cu extensie WIN32S fie pe Microsoft Windows NT.

SPIM poate citi si executa imediat fisiere scrise în limbaj de asamblare sau executabile MIPS. El contine un debugger si asigura câteva servicii specifice sistemelor de operare. SPIM este mult mai încet decât un calculator real (aproximativ 100 de ori). Totusi, costul scazut si larga aplicabilitate nu poate fi egalata de hardware-ul adevarat.

Microprocesor virtual, abreviat DLX, elaborat de catre profesorii John Hennessy (Univ. Stanford, SUA) si David Patterson (Univ. Berkeley, SUA), reprezinta o arhitectura RISC superscalara, didactica dar si performanta, cu cinci nivele pipeline de procesare, care este investigata prin intermediul unui excelent simulator software, înzestrat cu facilitati didactice deosebite si cu o grafica atragatoare. Scopul este de a ajuta la întelegerea conceptelor legate de procesarea pipeline a instructiunilor precum si a aspectelor arhitecturale specifice procesoarelor RISC. Pentru o întelegere mai buna dar si pentru a stârni interesul cititorului, în continuare se exemplifica sumar, pe diagrama ciclilor procesorului, desfasurarea procesarii pipeline a instructiunilor, aferente unui program de test.

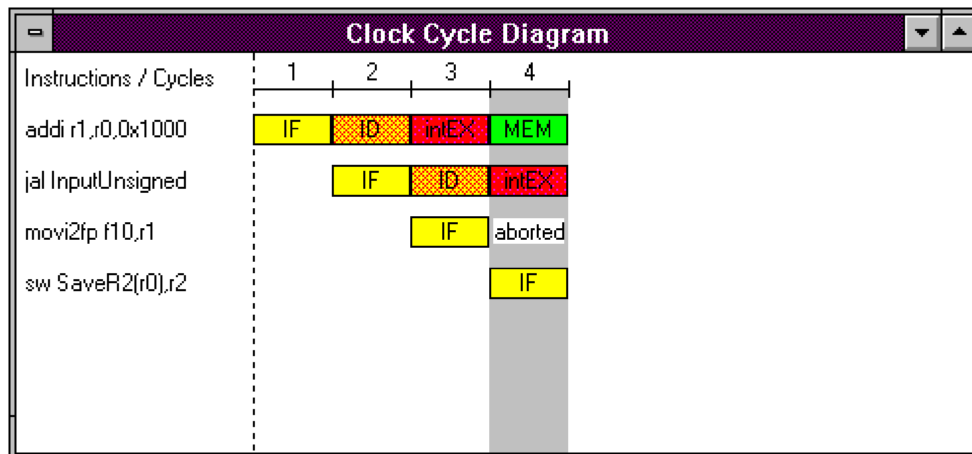


Figura 11.1. Diagrama ciclului de tact

În diagrama alaturata se observa ca simularea este în al patru-lea ciclu, prima instructiune este în nivelul MEM (acces la memorie), a doua în intEX (executie operatie aritmetico-logica) si a patra în IF (extragere instructiune din memorie). A treia instructiune este întrerupta. Motivatia consta în faptul ca a doua instructiune **jal,** este un salt neconditionat. Acest lucru e cunoscut abia dupa cel de-al treilea ciclu de tact, dupa ce instructiunea de salt a fost decodificata. În acest timp, instructiunea imediat urmatoare celei de salt (**movi2fp**) a trecut de faza IF, însa instructiunea care se va executa efectiv dupa instructiunea de salt se afla la alta adresa. Astfel, executia instructiunii **movi2f** trebuie întrerupta, lasând loc gol în pipe. Saltul se va face la eticheta *InputUnsigned*. Pentru detalii suplimentare accesati simulatorul DLX de pe CD.

Din grupul simulatoarelor de cache-uri amintim pe cel ce realizeaza simularea interfetei procesor - cache (BLWBCACH) într-o arhitectura superscalara parametrizabila. Scopul principal al simularii îl constituie evaluarea si optimizarea unor elemente esentiale – memoriile cache – ce caracterizeaza practic toate microprocesoarele superscalare avansate. Accentul este pus pe problematica cache-urilor, mai precis pe relatia dintre nucleul de executie si o arhitectura de memorie de tip Harvard, într-un context de procesor paralel, folosind tehnicile de scriere în cache cunoscute (*write back* si *write through*). Reamintim ca, o arhitectura Harvard se caracterizeaza atât prin spatii separate pentru instructiuni si date cât si prin busuri separate între procesor si memoria cache de date respectiv de instructiuni.

Al doilea simulator de cache-uri dezvoltat implementeaza conceptul de (*selective*) *victim cache* (HSVICTP2). Scopul urmarit este îmbunatatirea

performantelor memoriilor cache cu mapare directa, integrate într-un procesor paralel, utilizând conceptul de victime cache (simplu si selectiv). Pentru a reduce rata de miss a cache-urilor mapate direct (fara sa se afecteze timpul de hit sau penalitatea în caz de miss), Norman Jouppi a propus în 1990 conceptul de victim cache (o memorie mica complet asociativa, plasata între primul nivel de cache mapat direct si memoria principala; blocurile înlocuite din cache-ul principal datorita unui miss sunt temporar memorate în victim cache; daca sunt referite din nou înainte de a fi înlocuite din victim cache, ele pot fi extrase direct din victim cache cu o penalitate mai mica). Pentru reducerea numarului de interschimbari dintre cache-ul principal si victim cache, Stiliadis si Varma, au introdus în 1994 un nou concept numit selective victim cache(SVC). Investigatiile propuse se realizeaza cu ajutorul simulatoarelor amintite mai sus, parametrizabile, scrise în C, elaborate în cadrul grupului de cercetare în arhitecturi avansate de la Catedra de Calculatoare si Automatica a Facultatii de Inginerie din Sibiu.

Dintre cele doua predictoare de branch-uri prezente, primul (PAP) constituie o implementare detaliata a schemei de predictie cunoscuta în literatura de specialitate sub numele Pap (Per Address History Table, Per Address PHT) propusa de Yeh si Patt în 1992. Scopul simularii îl reprezinta investigarea unor arhitecturi moderne de predictoare a ramificatiilor de program (**branch**), în vederea reducerii penalitatilor introduse de instructiunile de ramificatie în procesoarele pipeline superscalare. Vor fi explorate în mod critic metodologiile de predictie existente, vor fi îmbunatatite, optimizate si stabilite limitarile fundamentale. Se vor stabili schemele de predictie optimale asociate diferitelor tipuri de ramificatii din program.

Cel de-al doilea simulator de acest gen reprezinta o varianta neuronala de predictor idee novatoare ce apartine unuia dintre autorii acestei carti. Predictorul neuronal de salturi este constituit în principal dintr-o retea neuronala, un registru de istorie globala (HRG) si o tabela cu registrii de istorie locala. Reteaua neuronala este o retea de tip Multi Layer Pertcepton având un singur nivel ascuns, un nivel de intrare si un nivel de iesire. Registrul de istorie globala (HRG) este un registru de deplasare de dimensiune parametrizabila si contine rezultatul ultimelor *k* salturi, având rolul unui prim nivel de istorie a salturilor. Tabela registrilor de istorie locala reprezinta al doilea nivel de istorie a salturilor. Spre deosebire de HRG care era un registru comun tuturor salturilor, în acest caz, fiecare PC are propriul registru de istorie. Privind dintr-o alta perspectiva putem considera predictorul format dintr-o banda de intrare (trace-ul ce va fi simulat), un automat de predictie (reteaua neuronala) si o memorie (HRG si

tabela HRL) ce contine istoria salturilor. Implementarea simulatorului este realizata în limbajul C++, iar ca mediu de dezvoltare Microsoft Visual C++ v.6.0. Fara a intra în amanunte legate de programarea sub Windows se poate spune ca structura interfetei utilizata este de tipul MDI, putând simula mai multe trace-uri simultan.

CD-ul însotitor cuprinde, de asemenea, doua simulatoare la nivel de executie a instructiunii, corespunzatoare arhitecturii superscalare concepute la Hatfield, Anglia. Primul, RES_SIM (simulator de resurse) realizat într-o forma primara, dar puternic parametrizabil, nu implementeaza conceptul de cache însa detine diverse alte facilitati precum: procesare in order, principalul rezultat generat fiind rata de procesare, vizualizarea gradului de utilizare a resurselor, generarea trace-ului de instructiuni în urma simularii etc.

Al doilea simulator aferent arhitecturii HSA (OUT_SIM) este mult mai complex si îl completeaza pe primul. Este implementat conceptul de cache, împreuna cu mecanisme auxiliare gen DWB (data write buffer) sau Outstanding Buffer (victim cache). De asemenea, noul simulator implementeaza mecanismul de executie *out of order* cu procesele componente corespunzatoare (*branch prediction*, *out of order instruction dispatch*, *renaming* aplicat registrilor prin algoritmi tip Tomasulo, *buffer de reordonare* pentru *mentinerea precisa a întreruperilor* si coerentei procesorului etc.).

Pe arhitectura superscalara HSA cu procesare in order s-a dezvoltat un scheduler (HSS) în scopul reorganizarii instructiunilor pentru eliminarea dependentelor date existente, eliminarea software a BDS (branch delay slot) si cresterea potentialului de paralelism. Schedulerul de instructiuni dezvolta o serie de tehnici: *software pipelining*, *combining*, *merging* (*move*, *immediate*), *inlining* aplicat procedurilor, analiza *anti-alias*, în scopul optimizari performantelor buclelor (basic-block-urilor) pentru cresterea globala a nivelului de paralelism disponibil. Daca pentru toate simulatoarele anterioare platforma de rulare era Windows '9x sau NT, platforma de executie pentru schedulerul de instructiuni HSS este Linux sau Unix.

Procesorul superscalar HSA încearca sa creasca paralelismul la nivelul instructiunilor atât **static** - în momentul compilarii - prin scheduling , cât si **dinamic -** în timpul executiei - prin diverse mecanisme (procesare out of order, utilizare DWB, branch prediction). Ambele simulatoare (RES_SIM si OUT_SIM) pot primi ca parametri de intrare fisiere cod masina schedulate sau neschedulate.

Un dosar (folder) va cuprinde programele de test (benchmark-urile) existente si necesare pentru simulare. Programele tip benchmark numite Stanford, sunt o suita de opt programe care necesita putina initializare de

date si care sunt gândite sa manifeste comportamente similare cu scopul general al programelor de calculator, desi unele au o natura recursiva. Programele implica executia a 100 pâna la 900 de mii de instructiuni dinamice. Codul original C este întâi trecut printr-un compilator *"gnu CC"* care produce formatul corect al codului în mnemonica de asamblare, precum si directive de asamblare, comenzi de alocare a datelor. Codul este apoi executat pe un simulator la nivel de instructiuni, care produce la iesire un fisier trace de instructiuni. De remarcat ca, la rularea repetata a aceluiasi program C aferent oricaruia din cele opt benchmark-uri se obtine acelasi fisier Trace. Fisierele prezente au urmatoarele extensii:

- ***.C*** - codul sursa original al programelor de test.
- ***.ins*** - varianta cod masina a benchmark-urilor necesare simulatorului de cache (BLWBCACH) si pentru cele doua simulatoare *execution driven* aferente arhitecturii HSA (RES_SIM si OUT_SIM).
- ***.inp*** - similare ca format cu cele *.ins* (practic o versiune mai noua), necesare ca parametri de intrare celor doua simulatoare *execution driven* aferente arhitecturii HSA (RES_SIM si OUT_SIM).
- ***.out*** - varianta schedulata (optimizata) a benchmark-urilor cu extensia *.ins*. Sunt utilizate de catre simulatorul RES_SIM pentru cuantificarea (determinarea) gradului de paralelism obtinut într-o procesare *in order* prin scheduling.
- ***.trc*** - fisiere trace reprezentând o înlantuire de triplete <*TipInstr AdrCrt AdrDest*>, unde *TipInstr* poate lua una din valorile 'B' – branch, 'L' – load, 'S' – store; *AdrCrt* reprezinta valoarea registrului PC – adresa instructiunii curente, iar *AdrDest* reprezinta adresa de memorie a datei accesate – în cazul instructiunilor cu referire la memorie ('L' / 'S') sau adresa destinatie a saltului – în cazul instructiunilor de salt si ramificatie ('B'). Sunt generate de simulatorul execution driven RES_SIM având ca parametri de intrare benchmark-urile .ins, si necesare simulatoarelor trace driven de cache-uri: (BLWBCACH, HSP2VICT). Exista totusi o deficienta a acestor trace-uri: nu evidentiaza salturile care nu se fac. Din acest motiv s-au generat noi trace-uri (fisierele *.tra).
- ***.txt*** - pentru simularea trace driven a interfetei procesor - cache într-o arhitectura superscalara parametrizabila (BLWBCACH), pe lânga cele 8 fisiere *trace* (*.trc) sunt necesare 8 fisiere identice ca nume având extensia (*.txt). Aceste fisiere (menite sa pastreze doar instructiunile), sunt o prelucrare proprie a programelor scrise în mnemonica de asamblare (*.ins), cu scopul de a ajuta la determinarea dependetelor de date reale existente între instructiuni (RAW) si aplicarea eventuala a unor tehnici de tip *forwarding* (*combining*).

- ***.tra*** - fisiere trace proprii, utilizate în cadrul predictoarelor de branch-uri implementate (PAP, NEURONAL). Sunt o prelucrare a programelor scrise în mnemonica de asamblare (*.ins) si a trace-urilor originale (*.trc), cu scopul de a evidentia toate salturile (inclusiv cele care nu se fac). Contin doar branch-urile (atât cele care se fac cât si cele care nu se fac) si exclud instructiunile Load / Store.
- ***.lib***, ***.dll*** - librarii statice si dinamice necesare în simularea arhitecturilor HSA.
- ***.shd*** - varianta schedulata (optimizata) a benchmark-urilor Stanford, utilizata de simulatorul OUT_SIM. Rezulta un fenomen foarte interesant si contrastant, si anume: performanta obtinuta (rata de procesare) printr-o executie out of order a fisierelor schedulate este inferioara celei obtinute la procesarea in order a fisierelor schedulate (si chiar celei obtinute prin procesare out of order a fisierelor neschedulate).
- ***.use*** - fisiere rezultat, de tip text, generate în urma simularii, la nivel de executie a benchmark-urilor Stanford, efectuata pe arhitectura HSA. Sunt utile analizei calitative si cantitative a performantelor obtinute. Cuprind rezultate semnificative: timp total executie, numar instructiuni procesate, grad de utilizare resurse, acuratete de predictie (acolo unde exista implementata o schema de predictie), configuratia arhitecturala aleasa, timp în care procesorul stagneaza, rate de hit în cache-uri (la simulatorul OUT_SIM).

11.2. PROGRAME DIVERSE

Sub acest titlu veti gasi o multime de programe utilitare din cele mai diverse domenii: viewer-e de documente în diverse formate, arhivatoare, browser de internet, kit de instalare Java, etc.

- Dosarul Adobe - kit de instalare pentru Adobe Acrobat Reader 4.05 (versiune pe 32 de biti) - citeste si afiseaza fisierele *.pdf* (freeware);
- Winzip.exe;
- Windows Commander v.4.5.1 (PC WORLD 9/2000);
- Java (TM) 2 SDK, Standard Edition, v.1.3 (PC WORLD 9/2000);
- Internet Explorer 5.01 - pentru Windows '9x si WindowsNT - (PC WORLD 9/2000);

11.3. DOCUMENTE

Trei dintre fisierele apartinând sectiunii curente reprezinta cartea de fata în format *.pdf*, *.doc* si *zip.*

- Cartea de fata în format *.pdf* sau *.doc* si arhivata cu utilitarul Winzip (format *.zip*).
- Alte fisiere în diverse formate *.html* si *.txt*, *.bmp*.

Obs. Toate programele cuprinse pe acest CD sunt shareware sau freeware si sunt incluse fara modificari. Autorii acestei carti nu-si asuma nici o responsabilitate pentru utilizarea acestor fisier si programe.

BIBLIOGRAFIE

---

[1] Philips Semiconductors – *80C51 – Based 8-Bit Microcontroller*, Integrated Circuits, Data Handbook, SUA, February 1994.

[2] Philips Semiconductors – *DS-750 Development Tools, User's Manual*, CEIBO, March 1994.

[3] Vintan, L. – *Arhitecturi de procesoare cu paralelism la nivelul instructiunilor*, Editura Academiei Române, ISBN 973-27-0734-8 Bucuresti, 2000.

[4] Chou, H., Chung, P. – *An Optimal Instruction Scheduler for Superscalar Processor*, IEEE Transaction on Parallelism and Distributed Systems, No. 3, 1995.

[5] Jouppi, N. – *Improving Direct-Mapped Cache Performance by the addition of a Small Fully Associative Cache and Prefetch Buffers*, Proc. 17^{th} International Symposium On Computer Architecture, 1990.

[6] Stiliadis, D., **Varma, A.** – *Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches*, TR UCSC-CRL-93-41, University of California, 1994 (republished in a shorter version in IEEE Trans. on Computers, May 1997).

[7] McFarling, S . – *Cache replacement with dynamic exclusion*, In Proc. 19^{th} Int'l. Symposium on Computer Architecture,1992, pages 192÷200.

[8] Hennessy, J., Patterson, D. – *Computer Architecture*: *A Quantitative Approach*, Morgan Kaufmann Publishers, 2-nd edition, 1996.

[9] Hill, M. – *A Case for Direct-Mapped Caches*, IEEE Computer, December 1988.

[10] Sugumar, R., Abraham, S. – *Efficient simulation of caches under optimal replacement with applications to miss characterization*, Performance Evaluation Review, 21(1):24-35, 1993.

[11] Belady, L. – *A study of replacement algorithms for a virtual-storage Computer*, IBM Systems Journal, 1966.

[12] Wall, D., Borg, A., Kessler, R. – *Generation and analysis of very long address traces*, In Proc. 17th Int'l. Symposium on Computer Architecture,1990, pages 290÷298.

[13] Steven, G.B. – *iHARP Instruction Set Specification*, Computer Science Technical Report No. 124, School of Information Sciences, University of Hertfordshire, June 1991.

[14] Collins, R. – *Developing A Simulator for the Hatfield Superscalar Processor*, Division of Computer Science, Technical Report No. 172, University of Hertfordshire, December 1993.

[15] Vintan L., Florea A. – *Sisteme cu microprocesoare. Aplicatii.* – Editura Universitatii "Lucian Blaga" Sibiu, 1999, ISBN 973-9410-46-4.

[16] Steven, G. B. – *A Novel Effective Address Calculation Mechanism for RISC Microprocessors*, ACM SIGARCH, *No 4*, 1991

[17] Collins, R., Steven, G.B. – *An Explictly Declared Delayed Branch Mechanism for a Superscalar Architecture, Microprocessing and Microprogramming*, vol.40, 1994

[18] Collins, R. – *Exploiting Instruction Level Paralelism in a Superscalar Architecture*, PhD Thesis, University of Hertfordshire, October 1995.

[19] Ebcioglu, K. – *A Compilation Technique for Software Pipelining of Loops with Conditional Jumps*, Proceedings of the 20th Annual Workshop on Microprogramming, ACM Press, 1987, pp 69-79.

[20] Ebcioglu, K., Groves, R. D., Kim, K., Silberman, G. M., Ziv, I. – *VLIW Compilation Techniques in a Superscalar Environment*, SigPlan94, Orlando, Florida, 1994, pp 36-48.

[21] Fisher, J. A. – *Trace Scheduling*: *A Technique for Global Microcode Compaction*, IEEE Transactions on Computers, C-30, (7), July 1981, pp 478-490.

[22] Lam, M. S. – *Software Pipelining*: *An Effective Scheduling Technique for VLIW Machines*, SIGPLAN 88 Conference of Programming Language Design and Implementation, Giorgia, USA, June 1988, pp 318-328.

[23] Nicolau, A. – *Uniform Parallelism Exploitation in Ordinary Programs*, Proceedings of the International Conference on Parallel Processing, August 1985, pp 614-618.

[24] Potter, R., Steven, G. B. – *Investigating the Limits of Fine-Grained Parallelism in a Statically Scheduled Superscalar Architecture*, 2nd International Euro-Par Conference Proceedings, vol. 2, Lyon, France, August 1996, pp 779-788.

[25] Rau, B. R. – *Iterative Modulo Scheduling*: *An Algorithm For Software Pipelining Loops*, Micro 27, November 1994, San Jose, California, pp 63-73.

[26] Rau, B. R., Fisher, J. A. – *Instruction – Level Parallel Processing*: *History, Overview and Perspective*, The Journal of Supercomputing, vol. 7, No. ½, 1993, pp 9-50.

[27] Sazeides, Y., Vassiliadis, S. – *The Performance Potential of Data Dependence Speculation & Collapsing*, IEEE Micro 29, 1996, pp 238-247.

[28] Steven, F. L. – *An Introduction to the Hatfield Superscalar Scheduler*, University of Hertfordshire, Technical Report No. 316, 1998.

[29] Steven, G. B., Collins, R. – *Instruction Scheduling for a Superscalar Architecture*, Proceedings of the 22nd Euromicro Conference, Prague, September 1996, pp 643-650.

[30] Steven, G. B., Christianson, D. B., Collins, R., Potter, R. D., Steven, F. L. – *A Superscalar Architecture to Exploit Instruction Level Parallelism*, Microprocessors and Microsystems, Vol. 20, No. 7, March 1997, pp 391-400.

[31] Steven, F. L., Potter, R. D., Steven, G. B., Vintan, L. – *Static Data Dependence Collapsing in a High – Performance Superscalar Processor*, 3rd International Conference on Massively Parallel Computing Systems, Colorado, April 1998.

[32] Warte, N. J., Mahlke, S. A., Hwu, W. W., Rau, B. R. – *Reverse If – Conversion*, SigPlan 93, Albuquerque, New Mexico, June 1993, pp 290-299.

[33] Vintan, L. – *Metode de Evaluare si Optimizare în Arhitecturile Paralele de tip I.L.P*, Editura Universitatii "Lucian Blaga" Sibiu, 1997.

[34] – *IA-64 Application Developer's Architecture Guide*, Intel Corporation, May 1999 (Order No 245188-001)

[35] Patterson, D., Sequin, C. – A VLSI Reduced Instruction Set Computer, Computer, October, 1982

[36] Smith, J., Sohi, G. – *The Microarchitecture of Superscalar Processors*, Technical Report, Madison University, August, 1995 (http://www.wisconsin.edu)

[37] Franklin, M. – *Multiscalar Processors*, Ph. D Thesis, University of Wisconsin, 1993.

[38] Patel, S. J. – *Critical Issues Regarding the Trace Cache Fetch Mechanism*, Technical Report, CSE-TR-335-97, University of Michigan, 1997.

[39] Maxim, C., Rochange, C., Sainrat, P. – *Instruction Reuse Based on Dependent Instruction Sequences*, In Proc. of 6th International Symposium on Automatic Control and Computer science, vol. 2, ISBN 973-9390-42-0, Iasi, 1998.

[40] Sodani, A., Sohi, G. S. - *Dynamic Instruction Reuse*, In Proc. of. 24th Annual International Symposium on Computer Architecture, pages 194-205, July 1997.

[41] Sodani, A., Sohi, G. S. – *An Empirical Analysis of Instruction Repetition*, In Proc. of. 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[42] Vassiliadis, S., Phillips, J., Blaner, B. – *Interlock Collapsing ALUs*, In IEEE Transactions on Computers, Vol. 42, No. 7, 1993, pp. 825 – 839.

[43] Lipasti, M. H., Wilkerson, C. B., Shen, J. P. – *Value Locality and Load Value Prediction*. In Proc. of. 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 138-147, September 1996.

[44] Richardson, S. E. – *Caching function results: Faster arithmetic by avoiding unnecessary computation*. Technical Report, Sun Microsystems Laboratories, 1992.

[45] Patt, Y. N., Yeh, T. Y. – *Two-level adaptive training branch prediction*, In Proc. of. 24th Annual International Symposium on Microarchitecture, pages 51-61, November 1991.

[46] Wang, K., Franklin, M. – *Highly Accurate Data Value Prediction using Hybrid Predictors*. In Proc. of. 30th Annual International Symposium on Microarchitecture, pages 281-290, December 1997.

[47] Sazeides, Y., Smith, J. E. – *The Predictability of data Values*, In Proc. of. 30th Annual International Symposium on Microarchitecture, pages 248-258, December 1997.

[48] Huang, J., Lilja, J. D. – *Exploiting Basic Block Value Locality with Block Reuse*, Technical Report HPPC, September, 1998.

[49] Sodani, A., Sohi, G. S. – *Understanding the Differences Between Value Prediction and Instruction Reuse*, In Proc. of. 31st Annual International Symposium on Microarchitecture, 1998.

[50] Tomasulo, R. – *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal, Nr. 11, 1967.

[51] *** – *Computer Review. The Future of Microprocessors*, IEEE Computer Society Press, September, 1997.

[52] Smith, J. E., Vajapeyam, S. – *Trace Processors*: *Moving to Fourth Generation Microarchitectures*, In IEEE Computer, september 1997, Volume 30, number 9, pages 68-74.

[53] Tsai, J. Y., Yew, P. C. – *Performance Study of a Concurrent Multithreaded Processor*, In Proc. of. 4^{th} International Symposium on High Performance Computer Architecture, Las Vegas, February 1998.

[54] PCWorld România , nr. 9/2000, Septembrie 2000, pg. 17, Editura Communications Publishing Group S.R.L., ISSN 1220-8639.

[55] Lepak, K., Lipasti, M. – On the Value Locality of Store Instructions, Int'l Conference ISCA 2000, Vancouver, 2000

[56] Williams, M. – *Bazele Visual C++ 4*, Editura Teora, Bucuresti, Octombrie 1998.

[57] Ryan, B. – *RISC Drives Power PC*, in Byte, August, 1993.

[58] Johnson M. – *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[59] Knuth, D. E. – *Tratat de programarea calculatoarelor*, vol. I, IV, Editura Tehnica, Bucuresti, 1974.